University of Southampton

AI Tools — Their Complexity And How To Manage It

by

Nigel Ian Payne

A thesis submitted for the degree of
Master of Philosophy

in the
Faculty of Engineering
Department of Electronics and Computer Science

November 1990

Contents

University of Southampton

ABSTRACT

Faculty of Engineering

Department of Electronics and Computer Science
Master of Philosophy

AI Tools — Their Complexity And How To Manage It

by Nigel Ian Payne

This thesis describes the difficulties encountered when using a particular AI tool to implement an Intelligent Knowledge-Based System (IKBS) to solve an apparently simple problem. KEE (Knowledge Engineering Environment) was chosen as a typical example of the more powerful, commercially available AI tools. It was found to be too complex: its different knowledge representation paradigms interacted in a complicated and, as a result, almost unpredictable fashion; it was difficult to control the reasoning process, especially where the production rules were involved. A subset of KEE's functionality was finally tamed by using Z, a formal specification language, to specify the problem and its implementation.

This work suggests that it is difficult for the programmer to be confident that an expert system or IKBS, built with this tool, would perform correctly under all circumstances. Moreover, it casts doubt upon the performance of applications built with AI tools of similar complexity. Further use of formal specification techniques during the design and implementation of these systems should increase their reliability.

Acknowledgements

I would like to thank my supervisor Professor D W Barron and also Professor P Henderson for their encouragement with my studies. I would also like to thank Dr D De Roure, Dr P Soper, Mr S Adams and Dr L Barroca for the many and varied discussions over the past four years. Mr S P Q Rahtz deserves praise for his efforts with Latex and Dr P H Lewis thanks for correcting proofs. Special thanks are due to Dr I R McNab for his support during times of woe. All of the above are or have been at the University of Southampton.

Finally, I would like to acknowledge the Science and Engineering Research Council who provided the grant for my research.

1 Introduction

The desire to understand the workings of the human brain has been with mankind for many years. Aristotle was the founder of Cognitive Science with his work 2300 years ago on knowledge representation. Leibniz, perhaps the first proponent of Artificial Intelligence (AI), attempted to quantify all knowledge and reasoning in his *Characteristica Universalis*. Today, research in this area is arranged into three major groups: Artificial Intelligence, Cognitive Science and Cognitive Psychology. AI has sprung from the discipline of Computer Science and approaches the problem of instilling intelligent behaviour within a computer. Cognitive Psychology has its roots in Psychology and investigates how people acquire knowledge, remember it, and put it to use to make decisions and solve problems. Cognitive Science is at the interface of Psychology and Computer Science: it emphasises the construction of a plausible model for the human cognitive processes which is then often implemented on a computer.

Since the 1950s there has been a growing interest in the field of Artificial Intelligence. 1956 saw 10 people, who would lead the field in the early years, assemble at the Dartmouth conference. Research continues today in both *symbolic* and *connectionist* (see chapter 2) approaches to AI. Despite the great promise of AI, it has proved more difficult to implement successful, large scale intelligent systems than the early pioneers of AI envisaged. This is probably not surprising given the time it took the human brain to evolve.

AI research is usually motivated by one of two goals (not necessarily mutually exclusive). One is to model the human mind in an effort to better understand it. The other is to construct a computer program to perform a task — often requiring intelligence — currently performed by a human.

Work associated with these goals may be classified as *strong* or *weak* AI. These terms were introduced by Searle in connection with his *Chinese-Room Thought Experiment* [see Searle 1980]. Proponents of strong AI believe that a computer running a program which models the human mind *is* a mind, in the sense that it can be deemed to *understand*. That is they adopt a *behavioural* stance on consciousness: there is no such thing as consciousness, there are only organisms behaving. Supporters of weak AI, on the other hand, would not credit the computer with *understanding*. They would instead see the computer as a powerful tool for studying the mind. Searle's work argues for the case of weak AI: he does not believe the computer can be deemed to be a *mind* with *understanding*.

The goal of using the computer for tasks usually performed by a human is common to both Computer Science and AI. In general the task may be tedious, repetitive, dangerous or difficult. AI applications would normally concentrate on tasks that require intelligence to perform. Such human expertise has always been valued. As technology advances, that expertise becomes more complex and thus its value increases accordingly. The ability to capture that expertise, in a computer program, would relieve the expert from the simpler of his tasks, allowing him to concentrate on the more taxing problems. It would also protect the employer, to some extent, against the loss of an expert and the associated knowledge.

The work presented here takes a pragmatic approach to AI: it is not concerned with arguments for or against strong AI but hopes to improve our ability to use the computer to implement successful AI programs (the meaning of *successful* is discussed in section 2.4).

The demand for AI software has resulted in several programming languages and environments becoming commercially available. This thesis discusses the difficulties experienced when applying an example of one of the larger tools, KEE (the Knowledge Engineering Environment), to a simple problem. The main difficulty was found to be managing the complexity of KEE. It is shown how a formal specification language, Z, may be used to specify the problem and its implementation. This helped to control KEE's complexity and resulted in a successful implementation within a reasonable amount of time.

The thesis is organised as follows:

Chapter 2: Knowledge Representation Knowledge representation (KR) is introduced via the notions of *modelling* and *abstraction*. The use of knowledge by human's is discussed briefly before concentrating on the issues involved with the successful representation of knowledge in computers. Several contemporary methods of representing knowledge are reviewed.

Chapter 3: Problem Solving and Implementation This chapter covers two areas. The first is the form of typical AI problems and strategies for their solution. The second is the AI programming environments (tools) currently available in which to implement a problem-solver. Discussion focuses upon KEE, as an example of a typical AI tool, in preparation for chapter 4.

Chapter 4: Applying KEE to a Design Problem The application of KEE to an example of a *design* problem, Alfa (see appendix B.2), is described. The difficulties encountered due to KEE's complexity are discussed. The use of formal specification (Z) in the implementation of a successful problem-solver is described.

Chapter 5: Future Work and Conclusions This chapter draws the thesis to a close and suggests how the work combining formal specification and AI tools should continue.

2 Knowledge Representation

2.1 Introduction

This chapter opens by identifying modelling and abstraction as key concepts associated with the process of knowledge representation (KR). It then briefly discusses the human use of knowledge before moving on to a more technical definition of knowledge itself. The problem of representing knowledge in computers is tackled and three critical topics identified, the KR language, the inference regime and the identification of domain knowledge (model). The principal KR formalisms used within AI are described. The chapter draws to a close with several issues pertinent to KR.

2.2 Modelling and Abstraction

> The notion of the *representation of knowledge* is at heart an easy one to understand. It simply has to do with writing down, in some language or communicative medium, descriptions or pictures that correspond in some salient way to the world or a state of the world. In Artificial Intelligence (AI), we are concerned with writing down descriptions of the world in such a way that an intelligent machine can come to new conclusions about its environment by formally manipulating these descriptions.
>
> Brachman & Levesque 1985

Obviously it would not be possible to write down everything that any one person knows about the world, thus only that knowledge that is relevant to the particular AI application should be presented to the computer. The computer will need sufficient knowledge about the problem domain to enable it to perform correctly. The quality of the AI application is often measured by performance alone, although the proponents of *strong* AI might argue that the end does not always justify the means. That is, the method that gives rise to the performance of the system, is just as important as the performance itself. In essence the programmer (or knowledge engineer) must construct a *model* of the problem domain. That is a simplified or idealized description of a particular system or process that is put forward as a basis for calculations, predictions or further investigation.

The process of identifying a simplified or idealized description is called *abstraction*. This is concerned with identifying the salient pieces of knowledge and representing them, hiding superficial detail. Only that knowledge that is used during the computer's reasoning process is needed, the rest is unnecessary. Thus, *abstractions* are formed for real world entities. For example, if the computer has to manipulate blocks on a table top, an abstraction for a block might identify features such as shape and weight to be important (see appendix B.1). If the computer has a means of sensing the colour of the blocks, then this too might be deemed relevant. However, if the computer's tasks never involved the use of colour to identify the block, it would be an unnecessary piece of information.

Each idealized description of an object, process, etc., is called an *abstraction*. A *model* usually refers to an abstraction of a large system of interacting entities or processes. Some abstractions will be for more complex objects than others. Thus there are different levels of abstraction. For example, a description of a car, at different levels of abstraction, may be something like:

- A *vehicle* is used for the transportation of people and objects.
- A *car* is a vehicle and consists of an engine, chassis, four-wheels and four seats.
- An *engine* consists of a cooling system, 4 cylinders, 4 spark plugs ...

So the abstraction of a car, hides the details of the engine. The description of the car could quite easily have included the details about the engine, in which case it would have been at a lower-level of abstraction.

This example also shows how abstractions may be used to *define* other abstractions. There are a variety of mechanisms for building up definitions, these will be discussed in section 2.9.3. These *definitions* may be used in both recognition of and reasoning about entities.

Knowledge Representation is concerned with developing both a medium in which to capture the knowledge, and a suitable model of the problem domain within that medium. The problem, presented to the computer, will often change as time progresses. It is therefore important that the KR medium should allow the model to be easily amended and extended. In terms of abstraction, this may be in the form of filling in previously omitted detail.

2.3 Human Use of Knowledge

So, what is knowledge? This section and the following will try to arrive at a useful definition of knowledge from the point of view of AI. This section discusses briefly the ways in which a person may be said to make use of knowledge. The next section offers a more technical definition.

Memories are one obvious form of knowledge. People can remember a variety of things: facts they have learnt; events in which they have taken part; decisions they have made, and so on. These type of memories can be classed as 'knowing what', they are a personal history. People also remember 'how' to do something — 'knowing how'. A standard example is knowing how to tie a shoelace. This is something the person was taught at one stage in their life, and have stored the knowledge for future use/re-use. This knowledge is *compiled* in some sense; it is not readily amenable to self-interrogation.

Man's ability to *communicate* has been seen as a major clue to understanding intelligence. It is also a skill which hinges on knowledge. Somehow people successfully manage to talk about the world around them, their feelings and opinions. They can issue orders or relate their own memories. Such a diverse skill must rely on converting some internal representation of their knowledge of the world into language [Jackendoff 1983].

Every moment of the day, a person is bombarded with sensory information. This input is decoded during the process of perception. Input from sight, sound, smell, touch and taste must be processed. Some inputs are recognised as known patterns and others are identified as questions requiring a response. It must be possible to store and recall all of these from memory. There are two key processes that have been identified with cognition: *recognition* and *recall*. Recognition is the process of identifying an incoming sensory pattern with one that has already been distinguished from others and stored in memory. Recall is the process of retrieving a particular piece of knowledge from memory, in answer to a question perhaps.

Humans also use knowledge to solve problems and play games. For example, planning the best order (best may be shortest route, for example) to visit shops on a shopping spree. This at least requires knowledge of the shops which may sell the items on the shopping list and of the town's geography. A game of chess involves knowledge of permissible moves and what constitutes a winning state. Past experience allows the player to select what is likely to be a winning strategy and more specifically a good move.

The following is a short list of different kinds of knowledge that it must be possible to represent:

General Knowledge: Language provides an insight into how humans store and manipulate knowledge: people speak about their personal knowledge of the world. There are different kinds of sentences, some appearing less ambiguous than others. It is often possible to construct different contexts which would radically change the precise meaning of a sentence. Here are some examples:

Simple Facts

John is a boy. Mary is married to John.

Does the second sentence refer to the same John as the first?

Rules

If the car's engine is running at 4000 revolutions per minute and the accelerator is still being depressed, then the computer controlled gear-box must change up a gear.

Beliefs

I think Mary is off work with a cold.

Ascribed Beliefs

I think John hates me because I lost his ball.

Procedural: People know how to carry out actions to achieve particular end results, e.g. tying a shoelace and playing the piano.

Images: It is possible to remember how scenes appeared to the viewer. There is also evidence for idealized mental images which can be used in reasoning. For example, when asked "Does a tiger have a striped tail?" many people report imagining a picture of a tiger and then exploding the tail's portion in order to *see* the answer [O'Shea & Eisenstadt 1984].

All these different types of knowledge need to be stored. Whether this is done by one or several means is still open to debate.

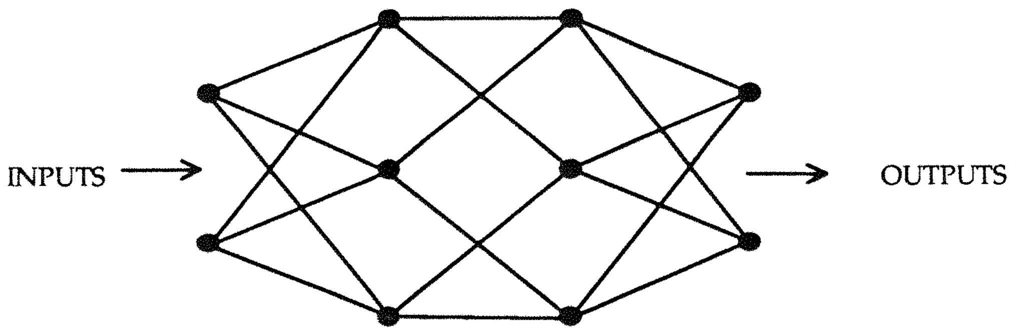


Figure 2.1: A net of nodes.

2.4 Definition of Knowledge

The Oxford English Dictionary gives the following definitions of the verb *know* and substantive noun *knowledge*:

> **Know** To perceive (a thing or person) as identical with one perceived before, or of which one has a previous notion; to recognise; to identify. Sometimes with *again*; also, later, with *for*.
>
> **Knowledge** Senses related to KNOWLEDGE (*verb*) and early uses of KNOW (*verb*). ...
>
> **5a** The fact of knowing a thing, state, etc., or (in general sense) a person; acquaintance; familiarity gained by experience. ...
>
> **8a** Acquaintance with a fact; perception, or certain information of, a fact or matter; state of being aware or informed; consciousness (of anything). The object is usually a proposition expressed or implied: e.g. the knowledge that a person is poor, knowledge of his poverty. ...

The definition of *Know* is in line with the use of knowledge in the process of perception and suggests the idea of a reservoir of knowledge with which to compare the perceived object. The later definition emphasises this notion of memory, 'knowing what'.

The phrase "familiarity gained by experience" refers to the learning process, and can equally be applied to 'knowing what' and 'knowing how'.

Current knowledge representations may be classified in many different ways. One particularly important division is between those representations which are predominantly *symbolic* and *connectionist* in nature. A *symbolic* knowledge representation is one that is based around the manipulation of symbols. In this case the symbol is a discrete uniquely identifiable token that is used to denote a particular abstraction from the problem domain.

The *connectionist* (or neural net) approach to knowledge representation normally has a network of nodes connected by links (see figure 2.1). Each node has a level of activation, α say, and a threshold of activation, θ (see figure 2.2). There may be two kinds of links, *excitatory* and *inhibitory*, both of which have weights associated with them. A node will *fire* when there is sufficient excitation on its input links. This

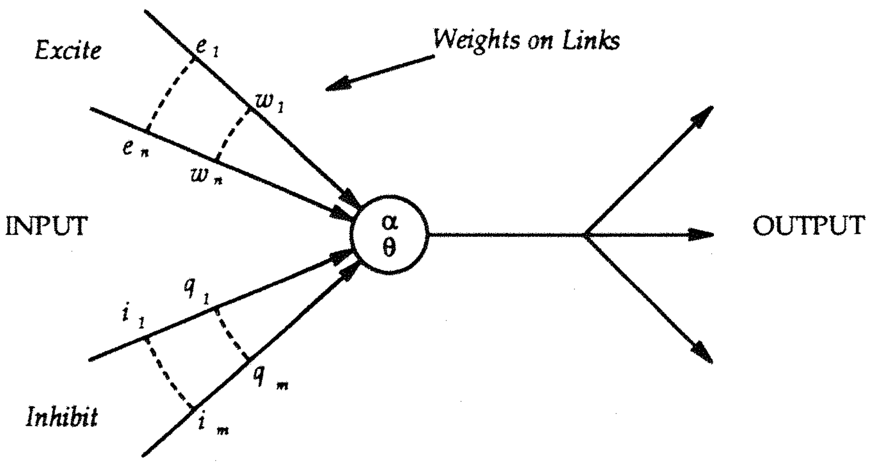


Figure 2.2: A single node and associated links.

provides a kind of *neural* inference mechanism. Information is stored in the net by means of adjusting the weights on the links or the threshold of the node. A single node (or perhaps a small group of nodes) represents an abstraction of the real world. These networks are used at widely varying levels of abstraction. Input to a node may be the gray level of a pixel in an image or it may represent something much more abstract such as a line in the image. Thus it is possible to have an AI system which uses both *symbolic* and *connectionist* knowledge representations at different stages in the reasoning process. For further discussion and references see Shapiro 1987.

Up until about 1984/85, the majority of AI had been concerned with symbolic rather than connectionist representation of knowledge. This is seen in Smith 1985 where he states the *knowledge representation hypothesis*:

> Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantical attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge.

Here he refers to "a propositional account" which clearly suggests a symbolic medium for representation.

This hypothesis also emphasises that the processes' intelligence (the computer plus program is the mechanical embodiment of the process) be judged by performance, much like the Turing Test [Turing 1950]. As discussed in section 1, Searle has argued that performance is *not* a suitable measure of intelligence [see Searle 1980].

Frost 1986 side-steps the question of connectionism by giving a definition of knowledge as used in the current knowledge-based system literature.

> Knowledge is the symbolic representation of aspects of some named universe of discourse.

This was probably an inherently *symbolic* definition because, to that date, most systems deemed to display intelligence (performance again!) had been symbolic in nature. More recently, connectionist systems have had success in application domains historically associated with AI, e.g. speech recognition systems. These connectionist systems are generally at a much lower conceptual level, being involved predominantly with low-level pattern matching rather than dealing with the high-level abstractions with which traditional AI is concerned.

It is important to distinguish *knowledge* from *data* and *information*. *Data* is a particular type of knowledge referring to simple aspects or facts of a particular universe of discourse. The *information* content of a piece of knowledge is governed by the hearer's prior knowledge: more information is conveyed if the hearer is being told something that they did not already know.

This thesis will also be concerned solely with symbolic knowledge representation. *Success* of the AI system will be judged upon performance, that is, how well the symbolic model of the domain performs under certain circumstances. The success of this model depends critically upon the abstractions chosen, and the ease with which they can be denoted and manipulated within the knowledge representation.

2.5 The Knowledge Representation Problem

Brachman & Levesque 1985 believe the central problem of AI (they also refer to this as the *representation problem*) has three components:

1. Selection and design of a knowledge representation language.
2. Inference regime.
3. Identification of particular domain knowledge needed to build a successful model.

2.5.1 Knowledge Representation Language

A knowledge representation language should be a *formal* language in which to record the *explicit* knowledge (see section 2.5.2) that the AI system contains. *Formal* means that the language should have well-defined *syntax* and *semantics*. In terms of our previous discussion of knowledge representation, the system of symbols (i.e. KR language) used to denote the abstractions of the problem domain, must be sufficiently well-defined that:

Syntax Legal structures (often called propositions or sentences) within that system can be distinguished from illegal ones;

Semantics Any reader of the notation can understand a unique meaning for any legal structure in the language, in terms of the abstractions of the real-world, and thus in terms of the real-world itself.

Such a KR language differs from those more typically used in computer science which only have a formal syntax e.g. Algol 60. These programming languages force the user to write legal structures but provide no rules for semantics, only english phrases. Therefore there is no guarantee that the resulting program will be meaningful and may well *crash* during execution.

2.5.2 Inference Regime

A knowledge representation has not only to be able to denote the abstractions of the problem domain but must also enable manipulation of those symbols (tokens) in an easy and correct manner.

Before proceeding further it is necessary to define two kinds of knowledge:

Explicit knowledge is the term for the knowledge within the AI system that can be seen to be present directly from the structures in the KR language.

Implicit knowledge refers to the knowledge that the AI system can infer from the *explicit* knowledge. The structures for this knowledge are not present in the AI system initially, but they can be constructed using well-defined manipulations.

It is precisely these manipulations that are known as the *inference mechanism(s).*

2.5.3 Identification of Domain Knowledge

As noted earlier, it is not possible to record all knowledge about any given domain. Instead, suitable *abstractions* must be identified for the domain and pieced together to form a *model.* It is always difficult to identify the precise boundaries of the domain, just as it is difficult to identify the correct level of abstraction at which to form the model. Often, enough knowledge will have been represented to cope with certain problems or situations that the AI system encounters, but when faced with a slightly different situation the system fails to perform as desired by the designer. It is at this stage, that the extensibility of the knowledge representation and model would be tested. Ideally an AI system should be able to cope with these unpredicted situations and adapt and extend the knowledge it already has.

2.6 Knowledge Representation Formalisms

A *formalism,* in this case, will often encompass not only a KR language but an associated inference regime as well. There are many such formalisms used in AI systems, including: subsets of natural language, conventional databases, formal logics, production rule systems, slot and filler systems, functional systems and programming languages. These will be discussed briefly below.

2.6.1 Natural Language

Natural language is used by humans to communicate a wide range of information. It can be used to represent what exists in the real world; what will exist; what might exist; what should exist and what one would like to exist, etc. Despite its power, it is often difficult for someone to express their feelings of emotion or the contents of works of art.

Natural language does not lend itself to computerization, for three reasons:

1. It is ambiguous, relying heavily on context to resolve the correct meaning of a sentence.

2. Its syntax and semantics are not completely specified.

3. It does not have a uniform structure: most successful techniques, for storing large quantities of knowledge, require the knowledge to be represented in a uniform format.

However research continues into the use of natural language for computer interfaces. Often only a subset of natural language is admissible, allowing the syntax and semantics to be well-defined.

2.6.2 Conventional Database Systems

Databases usually contain many simple facts stated explicitly, together with a few implicitly stored general rules. An IKBS would also contain explicitly stated general rules. For example, a system containing personal details may have a field for the individual's date of birth. The database system will ensure that this field is entered before the record is created. In an IKBS there would be an explicit rule indicating that every person must have a date of birth (expressed here in a formula of First Order Predicate Logic):

$$\forall x\, ((x \in \mathit{people}) \Rightarrow \exists y\, (x\ \mathit{hasdateofbirth}\ y))$$

2.6.3 Formal Logic

Languages of formal logic have well-defined syntax, semantics and rules of inference. Unfortunately they often have restricted domains of application and cannot represent all that can be said using natural language. Research is in progress to extend logical languages to cope with these shortcomings, e.g. non-monotonicity (see section 2.9.4).

2.6.4 Production Rules

A production rule based system consists of:

- A set of rules called production rules.
- A database management system.
- A rule interpreter.

A production rule is a *condition/action* pair, usually of the form:

If C then A

where C must be satisfied in the database (DB) before action A can occur.

The database management system deals with adding, deleting and editing facts in the database, etc. The rule interpreter is a program which identifies applicable rules and determines the order in which they are applied.

A production rule system stores specific knowledge as simple facts. These may be represented by any conventional DB techniques. General knowledge, however, is stored in the form of a set of rules. For example:

If X is a rabbit then it likes carrots.

where, X being a rabbit would be represented by a simple fact in the DB.

Some systems allow the user to associate probabilities with each rule. Thus if a group of rules is used to reason that the price of gold will rise in the next week, a probability may be calculated for this fact i.e. how likely the fact is to be true.

The advantages claimed of production rule systems are:

- They are modular. That is, they can easily be extended by adding extra rules.
- They can capture useful probabilistic or judgemental knowledge.

The experiments with KEE, reported in chapter 4, confirmed that production rules also have disadvantages:

- It easy to write a set of rules which apparently captures the knowledge necessary to solve a problem. However when these are used they fail or result in long periods of time spent searching various combinations of rules for the solution. In short, the reasoning process is hard to control.
- Following on from the above point, it is difficult to organise different levels of reasoning, i.e. at different levels of abstraction. Control is a form of higher level knowledge.

2.6.5 Slot and Filler Systems

These include frames, nets, conceptual dependency structures, conceptual graphs and scripts. They differ from many other representations in that they allow knowledge about a particular entity to be grouped together. Psychological arguments for doing this come from the notion of *association* of ideas. Technical arguments come from the ability to more easily access facts associated with an entity, if they are in a structure based around that entity. This kind of system is often easier for the human to use: the groups of facts about entities providing suitable abstractions with which the *user* can reason. Systems that have facts concerned with a single entity scattered throughout the knowledge base make it more difficult for the human to see the facts as all relating to one individual.

Slot and filler systems are often used for pattern recognition (patterns may be thought of as a group of facts); inference of generic properties, i.e. inheritance of properties from an entity describing the typical properties of an individual in that entity class; handling defaults; and detection of errors or omissions in larger bodies of knowledge. Their main disadvantage is that, due to the structure of the representation, the system can often have *ad hoc* inference rules.

2.6.6 Functional Approach

These systems regard the universe as a set of entities, with functions showing the relationships between those entities, e.g. a set of *students*, a set of *courses* and a function *course of* which maps *students* to *courses*.

2.6.7 Programming Languages

These are general purpose high-level languages for instructing the computer to carry out a set of operations. Until recently, these have been imperative and sequential in nature, however there are now languages that are more declarative and others that allow parallel processes to occur.

Programs in these languages suffer from being restricted to a particular, inflexible, application. It is often difficult to extend the application to cope with different problems/situations.

Two languages that are used widely within the AI community are Prolog and LISP. Prolog is a language with a declarative nature; it is based upon the idea of a logic database. LISP is a symbol-manipulation language where data and procedures have an equivalent form. Thus data structures may be constructed and then executed as procedures.

2.7 Inference

Inference is the name of the process which reasons from explicit to implicit knowledge, i.e. the conclusion of a new fact from information already known. There are several types of inference of which *deduction* is best understood.

Deduction This is a *logically correct* inference, that is, given true premises the deduction will result in true conclusions. For example,

From:	All cows eat grass. and
	Ermintrude is a cow.
Infer:	Ermintrude eats grass.

Abduction This generates plausible explanations, but is *not* a legal inference, i.e. the conclusion may be false. For example,

From:	All cows eat grass. and
	Dylan eats grass.
Infer:	Dylan is a cow.

However, Dylan might have been a rabbit! Thus abduction often requires an extra reasoning stage to select the most plausible explanation from several possible ones.

Induction Infers a general rule from several particular cases. Again this is *not* a legal inference. For example,

From:	Car 1 has four wheels. and
	Car 2 has four wheels.
Infer:	All cars have four wheels.

There are other types of inference. Larger knowledge representation structures, such as frames, often involve quite complex methods of inference. Some of these will be discussed later in section 2.9.3.

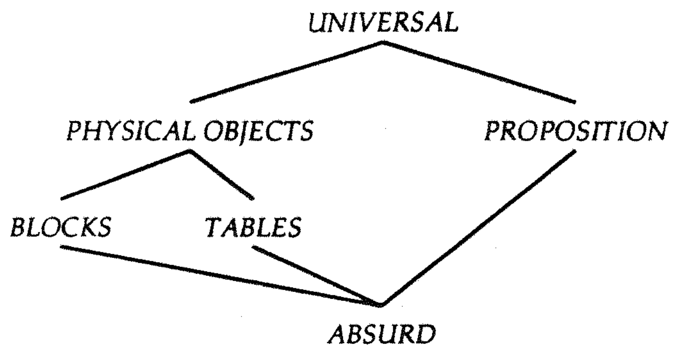


Figure 2.3: Concept hierarchy for Blocks World.

2.8 Analysis of Domain

Irrespective of the notation which is used to represent knowledge, it is always necessary for the 'representer' to perceive the universe and 'conceptualise' that part which he or she is wanting to represent.

Frost 1986

This process of *conceptual analysis* is critical to the success of the IKBS. Experience shows that this process can be very seductive and can lead to the consumption of valuable time. It is very easy to fall into the trap of trying to analyse the whole of the real world. Figure 2.3 shows a hierarchy (lattice) of concepts used during experiments with Sowa's conceptual graphs [Sowa 1984] and the blocks world problem (see appendices A and B.1).

An example of a simplified meaning for the hierarchy would be:

BLOCKS is a more specialised concept than *PHYSICAL OBJECTS.*

First attempts, at producing such an abstraction of the problem domain, involved many unnecessary concepts such as *DIRECTION, ABSTRACT OBJECTS, POSITION* and so on.

The key to successful *conceptual analysis* is to select the correct abstractions; defining only those that are necessary to pose the problem, reason and present the result.

2.9 Issues in Knowledge Representation

This section will discuss several important points so far untouched by this thesis.

2.9.1 Adequacy

A knowledge representation must be *expressive,* that is, the user should be able to represent any piece of domain knowledge he or she requires. However, the language should be kept as simple as possible, new structures only being added to the language where necessary. Indeed, unnecessary structures make the representation language more difficult for the user. For example, from Sowa's conceptual graphs, [Sowa & Way 1986]:

[PROPOSITION: λx
*[FARMER: *x] → (STAT) → [OWN] → (OBJ) → [DONKEY] ∀] –*
← (AGNT) ← [BEAT] ← (OBJ) ← [ENTITY: #]

This attempts to represent the following English sentence: *Every farmer who owns a donkey beats it.*

The representation should also allow *efficient* reasoning: it is no use asking for tomorrow's weather forecast if it takes 5 days for it to be calculated.

Both of these emphasise the need for the correct level of abstraction: if the language is too complex the user will find it difficult to read and write; if the descriptions within the language are too detailed the reasoning process will take too long.

2.9.2 Uncertainty

Not all knowledge is known to be true or false. There are many situations in which the knowledge may be incomplete, unreliable, vague and perhaps even inconsistent. In order to cope with these situations, the knowledge representation must be capable of showing such uncertainty in the truth of a fact or conclusion of an inference. In some cases *Probability Theory* from mathematics is sufficient. However, it is not always possible to suggest the *a priori* probability of the truth of a fact. This and other reasons have resulted in the construction of new theorems for uncertainty e.g. Certainty Theory [Shortliffe & Buchanan 1975] and the Dempster-Schafer theory of evidence [Schafer 1976], etc.

2.9.3 Intension and Extension

Some representations draw a distinction between the structures concerned with the description of the *type* of an individual, from those concerned with the individual's existence. Brachman & Schmolze 1985 call this *description* and *assertion*, others may say *definition* and *fact*. For simplicity, the knowledge base can be thought of as a collection of believed facts about *typed* individuals.

The *assertional* part of the knowledge base consists of a set of facts, declaring the existence and type of the individuals and the *relationships* that hold between them. The *descriptional* part of the knowledge base contains knowledge of what it means to be an individual of a particular type; this is in the form of a *description* or *definition*. For example,

Assertional:	There is an individual i1 of type BIRD. and There is an individual i2 of type PERCH. and The BIRD i1 is ON the PERCH i2.
Descriptional:	A typical BIRD will have a beak, two wings and a tail. A typical BIRD will be able to fly. ...

In this example, the descriptional part is trying to capture what it means to be a *BIRD*, i.e. the *intension* of *BIRD*. This intension may be used during reasoning processes about the individuals identified in the assertional part of the knowledge base. Here the *extension* of *BIRD* is *i1*. The third assertion declares that a *relationship* exists between the two individuals, namely, the bird is *ON* the perch. Some knowledge representations may also have given a description of this *relationship's* intension. For example,

Descriptional: The relationship ON may only exist between two individuals, one of which should be an ANIMAL
the other an INANIMATE_OBJECT.
And the ANIMAL can only be ON the INANIMATE_OBJECT, not the other way around.

Sowa 1984 has two kinds of *types*, namely

Concept Types Usually used for individuals often corresponding to *nouns* or *verbs* in natural language.

Relation Types Used for relationships between individuals.

Of these he concentrates more on *Concepts* than *Relations*. He discusses five ways in which a *Concept Type* might acquire its intension. These are discussed more thoroughly in Appendix A. Sowa fails, however, to define precisely how these different definitions are to be used in a mechanical reasoning process, thus making them next to useless.

One of Sowa's definitions is called a *Prototype*. This shows the form of a typical individual of the concept type. The graph specifies defaults that are true of a typical individual of this type, but *not* necessarily true for every individual. The above example describes a typical bird, however, a penguin is *not* a typical bird because it cannot fly. Thus the prototypical graph would say that a bird usually has two wings a beak and flies but in a specific case these may not be true. The means by which this is mechanised is already undergoing research.

2.9.4 Non-monotonic reasoning

Standard logics are *monotonic*: the set of all provable statements increase monotonically as new axioms are added. If the new axiom contradicts one of the previous axioms, the system becomes *inconsistent* and everything becomes provable. A logic, or any other knowledge representation formalism, is said to be non-monotonic if it can handle contradictory axioms without letting everything become provable. This may be achieved by restricting the inference mechanisms or by keeping contradictory facts apart.

The AI tool, KEE — Knowledge Engineering Environment, provides *Worlds* as a means of separating the inconsistent axioms. KEE will be discussed further in section 3.8.

2.9.5 The Frame Problem

A knowledge base will often be involved in reasoning about time, even if the time is not represented explicitly. Consider the blocks world example, one action follows another, although time itself is not expressly mentioned (see appendix B.1). As these actions occur, they are represented in the knowledge base by what is called a change of state. When a change of state occurs, new facts may be added to the knowledge base and old ones changed or deleted. At the same time, other facts will not change. Axioms expressing the facts that do not change between states are called *frame axioms*. The need to explicitly infer those facts that *do not* change between states is called the *frame problem*.

For example, consider the blocks world again, with three blocks on a table, a red, amber and green one. The robot arm then places the amber block on the green block. The changes that must occur within the knowledge base are:

- The amber block is no longer on the table.
- The amber block must be recorded as on the green block.
- The red and green blocks are still on the table.

The frame problem exists as a consequence of trying to model reality and thus causality.

2.9.6 Procedural Attachment

The models of the real world or problem domain, constructed within the knowledge representation, often take short cuts when it comes to sub-domains of which the computer may already have intrinsic *knowledge*. The most common of these sub-domains is *simple arithmetic*. When an IKBS needs to reason about arithmetic it will often make use of a function in a high-level programming language rather than explicitly representing mathematical knowledge. Charniak & McDermott 1985 state just this when talking about their *deductive retriever*:

> ... For instance, many mundane subgoals like
>
> ```
> (Show: (< 3 10))
> ```
>
> come up repeatedly. Rather than handle these with general axioms of the form
>
> ```
> (< number number)
> ```
>
> it just calls the LISP function < to do the test. This is called *procedural attachment*.

Sowa's conceptual graphs are particularly ill-defined when it comes to *numbers* and *sets*. Although Sowa mentions *procedural attachment* he does not say precisely how it should be implemented in conceptual graphs.

2.10 Summary

There are four important points that summarise this chapter:

1. The knowledge representation must have *well-defined*
    - Syntax
    - Semantics
    - Inference
2. The notation should be unambiguous to a human user. He or she must be able to read and write sentences, within the notation, which convey a single obvious meaning. If the user has any doubts as to the precise meaning of a sentence/structure, then the knowledge representation is of little use. This should follow from point 1.
3. A knowledge representation should deal both with *intension* and *extension*, using both during a reasoning process.

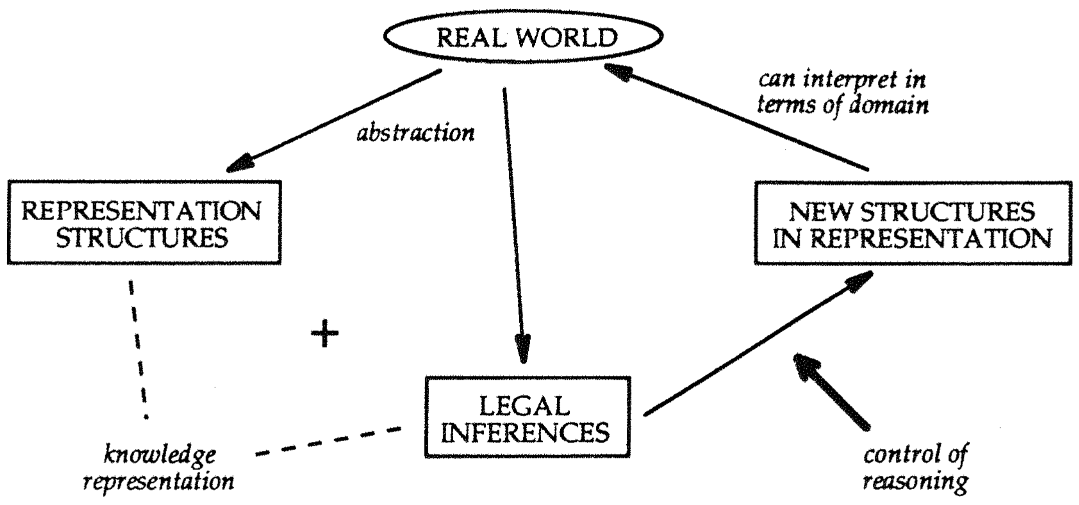

REAL WORLD
abstraction
can interpret in terms of domain
REPRESENTATION STRUCTURES
NEW STRUCTURES IN REPRESENTATION
+
LEGAL INFERENCES
knowledge representation
control of reasoning

Figure 2.4: The knowledge representation process.

4. The knowledge representation language should provide the necessary structures to allow the problem domain to be easily modelled. It should provide the ability to form abstractions at different levels. It must also be able to make use of those different levels of abstraction during reasoning.

The process of knowledge representation and reasoning are summed up in figure 2.4. This depicts the formation of the model by abstraction. New structures are created in the knowledge representation as a result of reasoning, using the legal inferences. The user may need to express some *control*, over that reasoning process, in order for it to be more efficient. The newly created structures are then interpreted in terms of the real-world.

3 Problem Solving and Implementation

3.1 Introduction

Chapter 3 opens by identifying the typical form taken by AI *problems* and their *solutions*. It then moves on to problem-solving strategies and their key components, *search* and *match*. The *generate and test* and *state-based search* strategies are described in detail, in readiness for the experiments reported in chapter 4.

The chapter then considers the process of implementation and the current AI tools or programming environments available. KEE, the *Knowledge Engineering Environment*, is chosen as a typical example of the more complex of these tools. KEE's features are discussed in detail, also in preparation for chapter 4.

3.2 What is a Problem?

Problems come in many different shapes and sizes: tying a shoe lace; diagnosing an illness; manipulating blocks on a table; playing chess; designing the layout of an integrated circuit; and so on. When the word *problem* is used in an AI context, it is usually referring to a specific task a computer should perform. Furthermore, the task probably involves some form of *reasoning* on the part of the computer: it will have been given a certain amount of knowledge about the problem domain and be expected to derive new knowledge in order to solve the problem. The *problem* may be thought of as the task of constructing a *bridge* from the *explicit* knowledge structures, the computer holds, to a *goal* structure. See figure 3.1.

The goal structure expresses *constraints* upon the solution to the problem. Goals may be constrained to different degrees. In some cases, all of the goal's details are constrained to particular values. In other cases, the user may not be worried about some of the goal's details, for example:

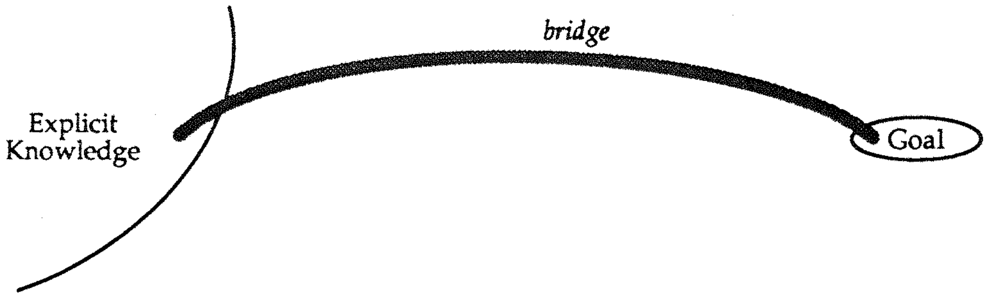


Figure 3.1: A bridge between explicit facts and the goal.

Build a tower consisting of 3 blocks with colours such that a red block is on an amber block which, in turn, is on a green block. However, I am not concerned about the shape of the blocks.

Thus the goal structure may contain *unknown* parts or both *unknown* and *known* parts; it is hoped that the unknown parts are *implicit* knowledge derivable from the *knowledge base*. A *knowledge base* is generally accepted as a collection of simple facts and general rules about a particular problem domain. The bridge is a record of how the *inference regime* uses explicit facts and legal inferences (often represented as production rules), to get between the *explicit* knowledge and the goal structure. That is the bridge is a record of how the goal is reached by the computer.

Problems often fall into one of two categories:

Synthesis: These are concerned with the construction of a goal structure from parts explicitly represented in the knowledge base. The goal structure usually represents something that the user wishes to pertain in the real world. These problems normally result in a *plan* containing the construction steps.

Analysis: These concentrate on explaining the goal structure in terms of the explicit knowledge represented within the knowledge base. The goal structure being given to the computer to represent something that pertains in the real world, in the hope that the computer can form an explanation in terms of explicit knowledge that it contains. This kind of problem usually results in an *explanation* recording its constituent steps.

Synthesis and analysis are just two convenient ways to classify problems.

3.3 What Constitutes a Solution?

Given the task of solving a particular problem the computer will produce one or more *outputs*. It is these *outputs* which form the *solution* to the problem. These *outputs* should be clearly identified before attempting an implementation of the problem-solver: it is often necessary to explicitly record the *transient* results of calculations because they form part of the specified *outputs*. Put another way, the *bridge* must be built for each problem but its precise structure will not always be stored explicitly.

Synthesis problems usually require the plan to be part of the required output. The details of the completed goal structure may also be important. Consider the construction of a tower of blocks again. Only the blocks which the tower contains may be of interest and not the plan. The plan would contain details of how blocks were moved around the table; these details may not be of interest whatsoever.

3.4 The Components of Problem Solving Strategies

A *problem solving strategy* is a method that the computer adopts in order to build the bridge. For example, a particularly optimistic builder might try to build a bridge by throwing the bricks into the air and hoping that they land in the shape of the desired bridge (perhaps not the best strategy!).

Before discussing the various strategies, it is necessary to discuss three activities intrinsic to *all* of them.

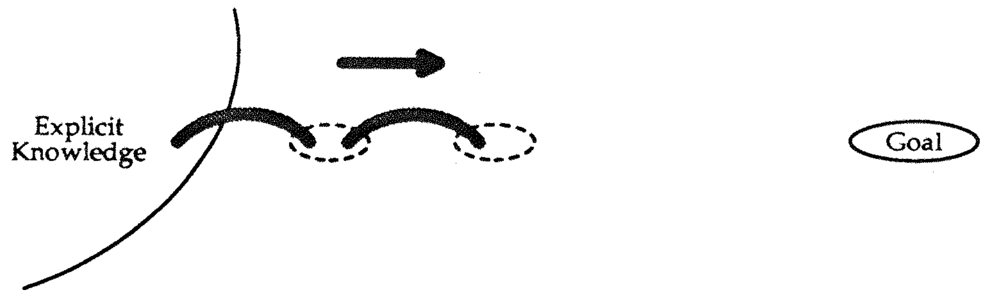


Figure 3.2: Forward search propagates from the *explicit* knowledge.

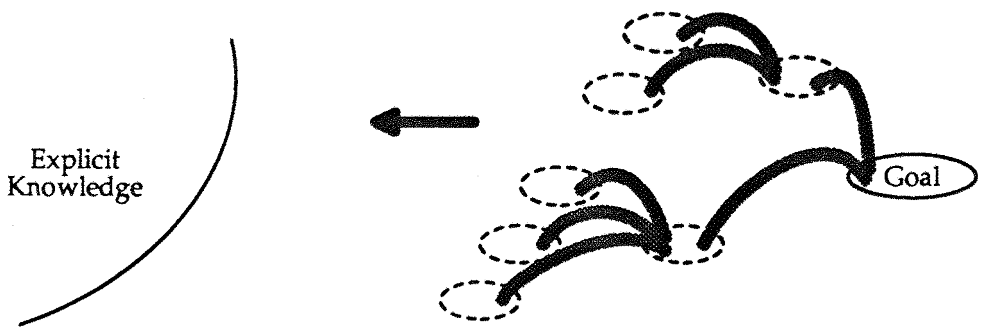


Figure 3.3: Backward search propagates from the *goal* structure.

3.4.1 Search

This is the process of trying different combinations of explicit facts and legal inferences in order to build the bridge. The different combinations give rise to what is called a *search space*. There are two basic kinds of search:

Forward: This is *data-driven* in nature. The computer starts to build the bridge from the explicit facts in the knowledge base. It tries the legal inferences, one after the other, when they are appropriate. In terms of the analogy above, the computer builds smaller bridges to intermediate islands, see figure 3.2. These intermediate islands correspond to implicit facts that the computer has inferred.

The computer is searching for a series of inferences that, when combined, constitute the complete bridge to a goal structure.

Backward: This is a similar process to the above, except that it starts from the goal structure. It is sometimes called *goal-driven*, the original goal being broken down into *sub-goals* until these are achieved trivially. Goals are achieved when they are explicitly present, as facts, in the system. See figure 3.3.

The *search* process frequently suffers from *combinatorial explosion* and, as a result, must be controlled in some fashion. Controlling search is a matter of current research. Algorithms, such as *Alpha-Beta* [see Winston 1984], may be used to prune the search tree. When the problem-solver is trying to reproduce an expert's reasoning process, *heuristics* may be used to reduce the search space or guide the search to an early result.

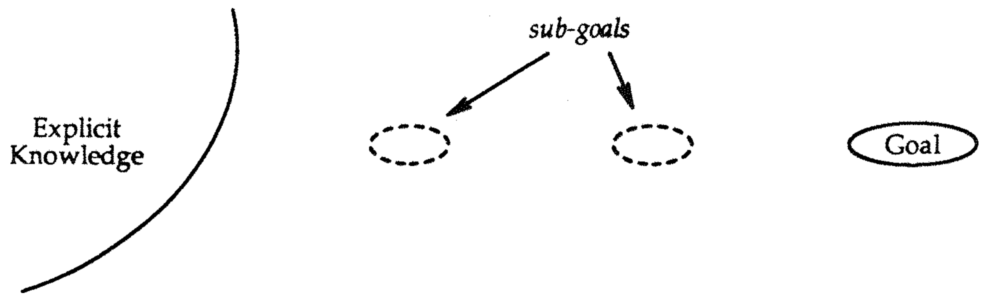


Figure 3.4: Identify sub-problems to reduce search.

These *heuristics* often capture *rules of thumb* used by an expert that may not be guaranteed to lead to the goal state. They might also represent theoretical knowledge (e.g. as contained in literature), such as mathematical algorithms.

Another method of reducing search, is to break the problem down into a sequence of *sub-problems* each having its own *sub-goal.* This divides the reasoning process into stages which must be achieved independent of whether the search is forward or backward. In the analogy, this would introduce islands that must have bridges built to them in order to bridge the gap posed by the whole problem. See figure 3.4.

Returning to the discussion of the classification of problems, note that synthesis is usually associated with a forward search from the explicit knowledge to the goal. Similarly, analysis is usually associated with a backward search, for an explanation, from the goal. This is not always the case: it is possible for backward search to produce a *plan* to be used for the forward construction of a goal structure.

3.4.2 Match

This is the process of identifying which legal inference can be applied. For example, in forward search using production rules, the *conditions* (or *premises*) of the rule must be matched against the facts in the knowledge base. Those rules with *matching conditions* are eligible for application.

Matching must also take place to detect when the goal structure has been reached during forward search and when the legal inferences, applied during backward search, have reached the explicit facts in the knowledge base.

3.4.3 Quantity, Quality and Efficiency

When specifying the *problem* it is necessary to answer several questions:

- How many solutions should the computer find? There is often more than one possible solution: one goal may give constraints within which several solutions lie.
- Are there any *measures of quality* for either the total or *partial* solution? What are they? What are their acceptable levels? Can they be used to guide search?
- Are there any *constraints* that may be used to reduce the search space?
- Is it necessary to find a solution quickly? Better solutions often take longer to find.

3.5 Different Problem Solving Strategies

There are many different strategies which may be adopted to attack a particular problem, those most commonly used are: state-based search; generate and test; describe and match; constraint propagation; means-ends analysis; and formal logic. When faced with a new problem the programmer (sometimes called a knowledge engineer, see section 3.7.3) must select different strategies that may be appropriate to the problem. The strategies should be seen as ingredients to the solution rather than a solution in itself [see Winston 1984].

Problem solving strategies are discussed in the majority of standard AI text books. Only two are discussed here, namely *generate and test* and *state-based search.*

3.5.1 Generate and Test

This method consists of principally two modules. The first is the *generator.* This has the task of *generating* possible solution knowledge structures for the problem. The second is the *tester* which, as its name suggests, *tests* the possible solutions, identifying those that *are* solutions.

These two modules may be combined in different fashions. The generator may produce a single possible-solution, pass it to the tester, and only produce another when the tester is ready. At the other extreme, the generator might produce *all* possible solutions before passing them to the tester. Moreover, the system as a whole may be content to find the first, several or all of the solutions, depending on the user's requirements.

In the terms of the analogy, this strategy builds the bridge in one step, rather than taking smaller steps, one after the other. Thus, in order for the system to be efficient, the generator must be *informed* so that it is disposed to create those possible-solutions that are more likely to be found correct. The generator must also be *complete*: it would generate *all* possible solutions, if it had to.

3.5.2 State-based Search

A *state* is an abstraction of part of the problem domain. It is a collection of variables (often called *state-variables*) whose values are intrinsically involved with the problem. The problem is expressed as an *initial* state, a goal state, and a set of *state operators.* The operators take a state and transform it into a new state. Knowledge that does not change during the search may be called the *context.*

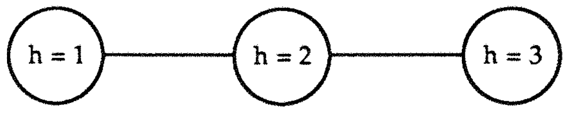


Figure 3.5: The states form a net.

Consider a trivial problem, that is suitable for solution using *state-based* search, consisting of a board with three pegs, labeled 1, 2 and 3. Initially there is a hoop on peg 1. The goal of the problem is to have the hoop on peg 3. However the hoop may only be moved between consecutive pegs e.g. from 1 to 2. This is the only *operator* in this problem, i.e. move the hoop to a neighbouring peg. There are thus only four legal

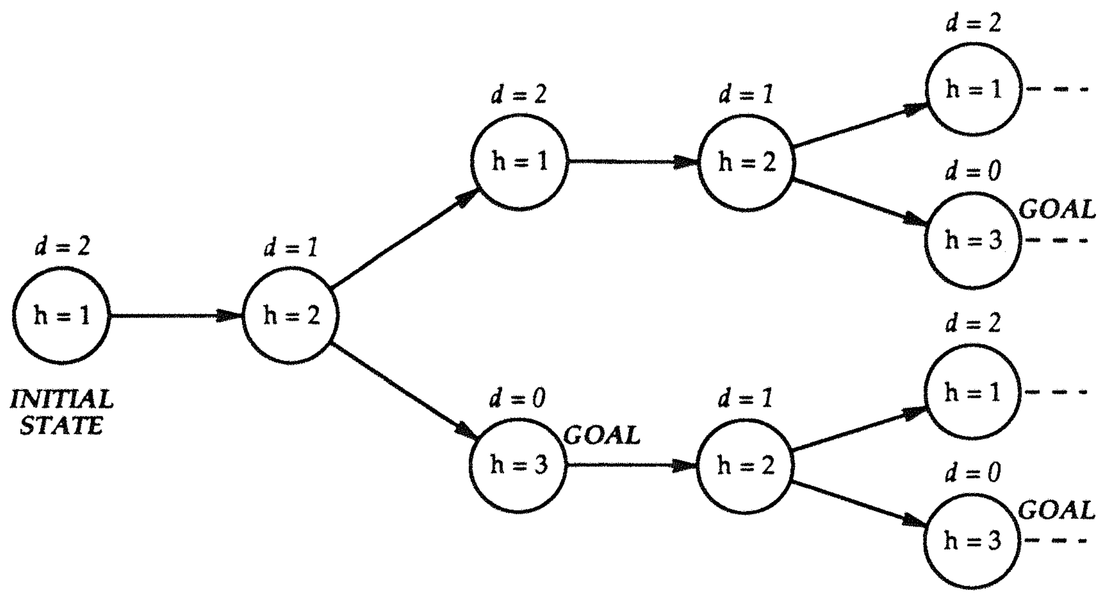


Figure 3.6: Searching the states forms a tree.

operations, move hoop from 1 to 2, move hoop from 2 to 1, etc. A state is represented by a state-variable, *h*, which may take the values 1, 2 or 3, depending on the position of the hoop. The possible transitions between states may be shown in a *net*, see figure 3.5.

The problem becomes the task of traversing this net successfully from the initial state to the goal state. The possible traversals transform the net into a *tree*, see figure 3.6. Starting from the initial state the computer must traverse the tree until it finds a goal state. The order in which the states of the tree are visited varies with different search algorithms. The three most common algorithms are described briefly below. The interested reader is referred to Winston 1984.

Depth-first: A branch of the tree is followed, as far as possible, before trying alternatives nearer the *root* (start) of the tree.

Breadth-first: All the states at a particular level of the tree are explored before considering the next level down.

Best-first: Given a choice of states to explore, a measure of quality is used to determine which looks most promising. An example of this is given below.

In this case, the best solution (smallest number of operations) the computer could find is to move the hoop from peg 1 to peg 2, and then onto peg 3. However, once the computer has moved the hoop to peg 2, it can then move to one of two states. If this choice is made arbitrarily, the computer may waste time searching in the wrong direction. A measure of quality for the partial solution can be introduced. This gives the computer an indication of how close it is from a solution state (goal state). In this simple example this measure might be defined as the difference between 3 and the peg number. Thus peg 1 is at a *distance (d)* of 2, and peg 2 at a distance of 1. By *looking ahead*, the computer can see that moving the hoop to peg 3, is the best move: 0 is less than 2.

3.6 Selecting a Strategy

The use of a problem solving strategy helps the programmer to break the problem down into manageable pieces by providing a skeleton upon which to cast the problem. This helps the programmer cope with the mental complexity of the problem. Moreover, the strategy will normally modularise the problem, thus making it easier to program a solution.

The strategy should be chosen on the basis of how easy it is to consider the problem in the terms of the strategy: state-based search would be ideal for a problem with a well defined set of operators and easily identified state-variables.

The knowledge representation used to describe the problem should provide the knowledge structures necessary to the problem solving strategy. For example, a classification problem would usually require some form of type-hierarchy in which typical examples of each type would be described.

3.7 AI Tools and Implementation

3.7.1 AI Tools

AI implementations take many forms due to the various knowledge representation formalisms. Some programs are extremely problem dependent and have a great deal of domain knowledge *hard-wired* into them. Many of the early AI systems were of this nature. This was soon seen to be a limitation and research started on systems that kept the domain knowledge and inference processes (reasoning) as separate as possible. It was hoped that this would enable the system to be applied to any problem domain with equal success.

Two terms have come into popular use, within AI, for applications constructed from knowledge representation tools that allow the separation of the domain knowledge from the inference mechanisms: *Intelligent Knowledge Based Systems* (IKBS) and *Expert Systems* (ES). There are various definitions for these terms; for the purposes of this thesis, they will be viewed as follows.

An Expert System is often implemented in an Expert System *shell* which usually has the following characteristics:

- Domain knowledge is represented in production rules. Initially there are no rules, hence the term *shell*.
- They have a friendly user interface. The rules and results are often presented to the user in a subset of natural language.
- They will be able to give some sort of explanation of their reasoning process: 'Rule 666: Turn off coolant, was triggered because: temperature T was greater than 1000°C; therefore: Coolant tap off.'
- They may possess the ability to calculate some form of probability associated with the result of their reasoning.

The user must program the system with the appropriate rules for the problem domain: Expert Systems are usually designed for a specific task, e.g. mineral prospecting. If probabilities are to be used, they must be associated with the rules. There are many

ES shells available at the moment, including ESP-Advisor (written in Prolog), Savoir and Xi (also in Prolog).

IKBS's are not so easily described, but in general:

- They are centered around a *knowledge base* (KB).
- They usually have the ability to maintain semantic integrity, unlike conventional database systems.
- They have the ability to use the knowledge in the KB to make inferences, thus allowing access to *implicit* as well as *explicit* facts.
- They can be used as general purpose sophisticated database systems or as components of systems with a specialised functionality, e.g. pattern recognition or playing chess.

There have been many languages and software development environments marketed to help with the construction of IKBS's. Amongst these are the *hybrid AI tools*. These are toolkits formed from useful AI and computer science programming techniques. They often have several different representation formalisms integrated within one system. Examples of such tools include: KEE — Knowledge Engineering Environment (written in LISP); Inference ART (also in LISP); Knowledgecraft; Leonardo; Flex (written in Prolog), KAPPA (written in C); KES and Egeria.

These toolkits contain various ingredients such as:

- *Frames* to represent structured objects.
- A hierarchy to allow inheritance of attributes and attribute values: both simple values and program code, often called *methods* (as in object oriented programming).
- *Demons* to watch particular attributes and carry out calculations in response to some trigger e.g. a request for the attribute value.
- *Production rules* which may have access to both simple and structured facts (frames) in the knowledge base.
- Graphical rule tracing facilities. These may be seen as giving some form of explanation of reasoning.
- A *truth-maintenance system* (TMS), to maintain dependent facts and provide explanations of their dependencies.
- A mechanism for segregating the knowledge base into parts in order to allow the exploration of alternative lines of reasoning.
- A *graphics toolkit* to help construct an interface, perhaps with windows, buttons, switches and thermometer displays etc.
- The user is often allowed access to the underlying programming language which is frequently LISP or Prolog.

3.7.2 Knowledge Elicitation

When implementing an AI system, as with any computer system, it is important to reach a clear specification of the requirements. This means that the particular problem, that the system should solve, must be described as clearly as possible. A major step towards achieving this, is to identify the *inputs* and *outputs* that the system is expected to handle.

One of the most important stages in building an AI system, is the identification of the domain knowledge with which the computer should be armed. In some cases there will be an *expert* in solving the problem, in others the knowledge must be drawn from books and sometimes the programmer's own experience. *Knowledge acquisition* is the process of acquiring information concerned with solving the problem from *any* source: text books, human experts, etc. *Knowledge elicitation* is a sub-set of this process, and is restricted to eliciting knowledge from a human expert.

There are several popular elicitation techniques including: structured interviews; protocol analysis; concept sorting; laddered grids; the limited information task (otherwise known as 20 questions); and automatic elicitation techniques. Shadbolt & Burton 1989 describe these in detail.

AI researchers often consider simplistic *toy problems* in order to investigate the knowledge needed to solve a problem. This is done in the hope that the lessons learnt, from the simple case, may be generalised and applied to more complex real-world domains. In these situations, the researcher often deems himself to be the expert in solving the problem. Obviously this is considerably different from trying to elicit knowledge from a busy expert whose time is costly.

3.7.3 Implementation

Once the problem solving knowledge has been elicited, it is necessary to identify the abstractions needed to form a suitable model of the problem domain. The problem solving strategies involved must also be isolated. At some point, the knowledge representation formalism must be chosen. Because the knowledge elicited may be more easily represented in one formalism than another, it may affect this choice. The person responsible for this process has become known as a *knowledge engineer*.

Having captured the knowledge in a programming language, AI tool, expert system shell, or other knowledge representation system, the implementation must be tested on several test cases. This will normally lead to the revision of the implementation and re-testing. This continues until the knowledge engineer and expert (if one exists) are satisfied with the performance of the system. The system is then assumed to be working correctly for the problem domain, although further revision may be necessary at a later date. This differs from the current trends in conventional computer science where efforts are made to prove that the program can cope with *all* the situations it will meet. Due to the nature of most IKBSs, it is unlikely to be possible to prove that the system will perform correctly under *all* circumstances. There is, however, some work currently in progress which explores the use of formal specification in the development of AI software. McCluskey 1988 reports the use of formal specification during the design and implementation of a correct logic program for a non-linear planner.

3.8 KEE — Knowledge Engineering Environment

KEE was chosen as a typical example of a hybrid AI tool: it contains many of the features present in other AI tools; it was readily available; and it was a tool with which I had had several months programming experience. It is described in detail below in preparation for the experiments which are reported in chapter 4.

KEE was developed by IntelliCorp Inc. It is intended as an environment for representing and reasoning with knowledge, for the development of large AI applications. KEE was first introduced in August 1983 and is currently available as Version 3.1. It is written in Common Lisp (the interested reader is referred to Steele 1984) and may be used on computers from LISP Machines Inc., Sun Microsystems, Symbolics, Texas Instruments, Xerox, Apollo, DEC (AI VAXstation), Hewlett Packard and IBM (PC-RT).

KEE is a very large system; in order to run it on a Sun workstation, for example, it requires:

- At least 7 Mbytes of disk file space to load the files from tape.
- Greater than 18 Mbytes to install and configure.
- Greater than 50 Mbytes of disk swap space.
- At least 12 Mbytes of RAM installed (16 Mbytes recommended).

Due to its size and complexity, it takes several months to learn the essential components of KEE. Many months are required before the user is really competent. From my experience, it is also necessary to have a working knowledge of LISP: although KEE provides high-level knowledge representations, LISP is often used within those representations, e.g. the production rules.

KEE is a true hybrid of several high-level inference and knowledge-base management facilities built upon Common Lisp. These are as follows:

- A frame-based representation of objects and their properties.
- *TellAndAsk*, a user friendly interface based upon a subset of natural language.
- *Rulesystem3*, a production rule system.
- An assumption-based truth maintenance system (ATMS).
- *KEEworlds*, which allows alternative reasoning paths to be explored simultaneously.
- *ActiveImages*, which allows the display and adjustment of a slot's value by a predefined graphics interface.
- *KEEpictures* is also used to construct a graphics interface. It provides more primitive commands than ActiveImages, for windows and drawing etc.
- A suite of debugging aids.

3.8.1 Frame-based — The Class Hierarchy

KEE represents *objects* and their *properties* by *units* and *slots* respectively. *Unit* is the KEE term for a *frame*. *Relationships* between objects are usually represented by a slot although it is possible to model relationships, in a more complicated fashion, using units. There are two kinds of unit, a *class* and an *instance* unit. The former represents a type of individual, the latter a particular instance of that type.

Classes are organised into a hierarchy. This saves re-representing knowledge (re-coding) by utilising *inheritance* down the hierarchy. The hierarchy also helps the user conceptualise the problem domain: object types in the domain map directly onto classes in the hierarchy; the classes provide a way of abstracting the appropriate detail from the domain.

Each class may have associated with it two types of slot, called *own* and *member* slots. *Member* slots are inherited by any unit beneath the class in the hierarchy. *Own* slots are not. A slot can contain a simple value — a string or number. It may also hold more complex values, such as another *unit* in the hierarchy or a *method*. A *method* is basically a piece of LISP code which is applied when the appropriate *message* is sent to the *unit*. This code may carry out a simple calculation, change the values of another slot, invoke some of the KEE commands, assert facts into the knowledge base or send a *message* to another *unit* in the knowledge base. The following is an example of the syntax used for sending a message to a *unit*:

```
(unitmsg 'car-1 'calculate-petrol-consumption)
```

There are two special operators, `before` and `after`, which allow methods to be combined with those inherited from parent classes. The method code defined locally is evaluated *before* or *after* the inherited code depending on this operator.

Each slot may have a series of *facets* associated with it. A *facet* would most typically contain an *active image*, *default* value or a *constraint* on the value that the slot may hold. For example, the value of a slot might be confined to be an instance of a particular class in the knowledge base. The facet that fulfills this purpose is called the *valueclass* of the slot. Users may also define their own facets.

3.8.2 TellAndAsk

This provides an interface to the knowledge base using a subset of natural language in a structured template. There are four permitted templates:

```
(the <slot> of <unit> is <value>)
(the <slot> of all <unit> is <value>)
(<member unit> is in class <class unit>)
(all <subclass unit> are <superclass unit>)
```

Three TellAndAsk functions allow the user to modify or query the knowledge base:

assert: This adds a fact to the knowledge base.

retract: This deletes a fact from the knowledge base.

query: Variables may be placed in the templates and thus used to *retrieve* data. For example, to find the **`colour`** of **`car-1`**:

```
(query '(the colour of car-1 is ?value))
```

The four templates may also be used in the definition of rules:

```
(if (the temperature of ?thing is hot)
 then
    (?thing is in class hot-things))
```

A fourth TellAndAsk function, `ask.user` is used to prompt the user for information. If there are no variables in the template, the system is asking the user if the fact is true. If there are variables, the system is asking the user for all the true instances of the variables. Any new facts are asserted into the knowledge base.

Apart from the four templates, which are used to manipulate *structured* facts, the TellAndAsk functions may also be applied to text or *unstructured* facts. For example:

```
(assert '(text (backwards speak us of all today)))
```

This says that the fact "backwards speak us of all today" is true. Thus the following query would return true:

```
(query '(backwards speak us of all today))
```

3.8.3 Rulesystem3 and the Truth Maintenance System

Rules may be grouped in KEE using *rule classes* and are created as instances of a particular rule class using an editor. When rules are invoked, chaining can be restricted to a particular rule class. This helps the programmer control the inference process. KEE allows forward, backward and mixed chaining of rules. KEE's rulesystem mechanism is very complex; the following is a brief summary of its activity. It is not necessary for the reader to understand the intricacies of this mechanism but they should have an impression of its complexity.

Forward Chaining When a fact, F, is asserted and KEE told to forward (FW) chain using a particular rule class, RC say, it:

1. Finds the rules in RC with premises that match F. These rules are said to be *tickled.*
2. The tickled rules then have the rest of their premises checked. Those that have a complete set of true premises are placed on an *agenda.* This occurs in the order that rules are displayed in a rule class, i.e. alphabetical. There are different types of rule: *deduction, same world action* and *new world action.*
3. As the rules are added to the agenda they are ordered, first by rule type (order as introduced above). Within each type, rules are ordered in a depth first fashion (default).
4. The programmer can customise an *agenda controller* that orders the rules on the agenda. This can be done for both the addition and selection of a rule to and from the agenda.
5. When no more rules are tickled, and they have all been scheduled on the agenda, the rule at the top of the agenda is selected. It may then have its premises rechecked; this depends on a global flag set by the programmer — *verify rule premise.*
6. Once a rule has been successfully selected from the agenda, it has its conclusions asserted. This may cause more rules to be tickled. It may also swap chaining to another rule class.

7. Chaining may occur in a breadth-first or depth-first fashion, depending on how the initial fact F is asserted. If breadth-first, the tickled rules are placed on the bottom of those in the corresponding rule type, previously on the agenda. If depth-first, then they are placed at the top of the corresponding rule type on the agenda.

Backward Chaining This is invoked with a *query* for a particular fact F, perhaps using a rule class RC.

1. The BW chainer works by growing a derivation tree containing *derivation nodes*. These nodes are stored on an agenda. Each node has an associated *goalstack*. When the goalstack for a node is empty, the problem represented by the node is solved.
2. A rule is searched for with fact F as a *conclusion*.
3. The premises of the rule are placed on the first node's *goalstack* in a depth-first fashion (the default, this can be changed).
4. The first premise is the first *goal* to be taken from the *goalstack*. It is searched for, as an explicit fact, in the knowledge base.
5. If it cannot be found as an explicit fact and there is another rule, within the rule class, with it as a conclusion, BW chaining occurs. A new node is created with new premises being placed on the new goalstack.
6. A node is solved when it has no goalstack: this is when the goals have been found as explicit facts within the knowledge base or child nodes have been created, via rules, and they have been solved.
7. The agenda of nodes is ordered first by rule type (as FW chaining). The BW chainer can also be instructed, by the programmer, to order the rules by *premise complexity*, *weight* or by use of a programmer defined function.
8. The derivation is explored in one of three ways: depth-first, breadth-first or best-first. *Best*, in this context, means smallest goalstack.

The three rule types mentioned are *simply* explained as follows:

Deduction: These have the general form:

```
(while <premises> believe <conclusions>)
```

They cause the creation of *justifications* in the Truth Maintenance System (TMS). Such justifications may be set up to show dependencies between *simple* facts (sometimes called TMS facts). A *simple* fact, in KEE, is one that is either the value of an *own* slot, an *unstructured* fact or a special fact, `false`. For more details about assumption-based truth maintenance systems, the reader is referred to deKleer 1986. The premises of a deduction rule may be true in any *world* or the *background*.

Same World Action Rules: These have the form:

```
(if <premises> then <conclusions>)
```

As there name suggests, their premises must be true in a single *world* (see below); this is also where the *conclusions* are asserted.

New World Action Rules: These have the form:

```
(if <premises> then in.new.world <conclusions>)
```

If FW chaining the premises may be true in one or several *worlds*. If BW chaining they must be true in the same *world*. In FW chaining, the rules are only applied to the *highest* (most recent) *worlds* and the conclusions are asserted in a newly created *world*.

So, as the reader will appreciate, KEE's *Rulesystem3* is very complex, and as a result very hard to learn and control (difficulties encountered with this rulesystem are described in section 4.7).

3.8.4 KEEworlds

Worlds may be used to represent alternative states of knowledge. They provide a way of grouping together facts that are only true in that world. For example, consider planning the construction of a tower of cubes in the blocks world (see appendix B.1). Let there initially be a red and blue cube, both of which are on a table. If the system has a rule that allows it to place one cube on top of another, two possible situations could result. Situation 1, would have the red cube on the blue cube, and the blue cube on the table. Situation 2, would have the position of the red and blue cubes reversed. Because both situations cannot be true at the same time, the rule must be a *new world action* rule. Thus, in order to consider the alternative situations, two worlds would be created to separate the inconsistent facts.

The *simple* TMS facts are the only type of fact that may vary between worlds; it is not possible, for example, to change the organisation of the class hierarchy within different worlds. Such knowledge is held in the *background* context and will be the same in *all* worlds.

There are many uses for KEEworlds, including:

- Exploration of the affects of different values of *own* slots on the reasoning process.
- Rules may be run solely within a world, without changing the original state (background context) of the knowledge base.
- They may be used to record stages in a reasoning process.
- They can be used to explore different hypothetical states of the knowledge base.

Thus worlds are often used for problems involving planning, configuring or exploration of alternatives.

3.8.5 KEEpictures and ActiveImages

KEE provides a toolkit — *KEEpictures* — for manipulating windows and general graphics, e.g. bitmaps. This was not used during the experiments described in the next chapter and does not need to be discussed further.

ActiveImages, which is built upon KEEpictures, was used to provide a means of rapidly prototyping interfaces and testing the IKBS, as well as constructing the final interface. An ActiveImage may be associated with a particular slot, of an instance in the knowledge base, to provide a graphical display of its value. Some ActiveImages also allow the user to change the value of the slot using the *mouse*, e.g. toggle switches.

3.9 Summary

There are four important points that summarise this chapter:

1. AI problems are usually concerned with inferring *implicit* knowledge from a given amount of *explicit* knowledge about the problem domain.
2. It is important to identify the specific outputs that form the solution.
3. There are different problem-solving strategies which may be appropriate components of the overall strategy best suited to the problem in hand.
4. KEE is selected as a typical example of the hybrid AI tools, currently available commercially, for the construction of problem-solvers.

4 Applying KEE to a Design Problem

4.1 Introduction

This chapter reports the application of KEE to an apparently simple design problem. Problem-solvers built in LISP and Prolog are also described to allow comparison with KEE. The difficulties encountered with KEE, mostly due to its complexity, are discussed. Finally the successful application of formal specification to this problem is described.

4.2 The Problem

The design of new systems from a number of smaller systems (or components) is a problem inherent in many situations. One of particular interest is that of designing the architecture of a computer system — system architecture. The system architect (SA) must combine several components to form a system to meet some form of specified requirements. He may also be asked to suggest the components and combinations a company should be developing and marketing for the future.

Henderson and Warboys have recently been developing a computer language (Alfa) to help the SA with his job [see Henderson & Warboys 1989a, b, and c]. The basic idea behind the language is that it should provide a playpen for the SA to experiment with different combinations of components. The SA might be looking for a combination to meet a given specification, or perhaps experimenting to see what a particular combination provides. The Alfa language is one in which the SA describes the structure of the system he is exploring in a declarative fashion. He then probes the structure to discover which functions it provides.

The work presented here has taken the lead from Alfa but encompasses a different philosophy of use. Rather than the SA constructing the structure in which he is interested, he instead provides the computer with a database of primitive components and systems built from those components. He then specifies the requirements of the desired system. The computer attempts to construct a system from its database of components and previously constructed systems (a system can contain sub-systems) to meet the specification. The SA might request the best possible solution according to some criteria, or be content with any one solution, or perhaps want all possible solutions to be listed. This problem falls within the *synthesis* problem category discussed in section 3.2.

4.2.1 Notation

The notation presented here is not exactly the same as that used in Alfa: implementation of the IKBS led to a greater understanding of the concepts put forward in Alfa. The following concepts will be used throughout the implementations and discussions.

Services These are a primitive type of object possessing nothing more than a name.

Modules A *module* is a primitive, non-decomposable component. Modules *require* and *supply* a set of services. The SA is expected to declare a group of primitive modules to be part of the database that the IKBS can draw upon to form new components.

Systems These are decomposable components. In addition to having both supplied and required services, a system also has a number of *hidden* services. It is also composed from a number of other systems and thus has a number of *components*. A module can be thought of as a system with no components and no hidden services.

Structure A *structure* is the name given to a subset of the knowledge about a system. It is knowledge of both the components of the system and the hidden services.

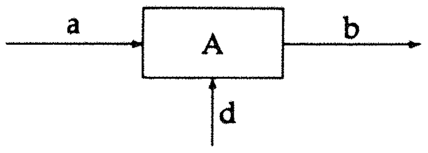
The specification of a desired system This is another subset of the knowledge about a system. A system may be specified as needing to require and supply services. On some less frequent occasions, the hidden services may also be specified.

Component This term is used to refer to modules, structures and systems.

Alfa itself has both a linear and more evocative pictorial form. These are used in appendix B.2 to present example problems that will be referred to later. The following are examples of the Alfa language:

A is a module that requires services a and d and supplies a service b.

A = **unit requires** a, d **supplies** b

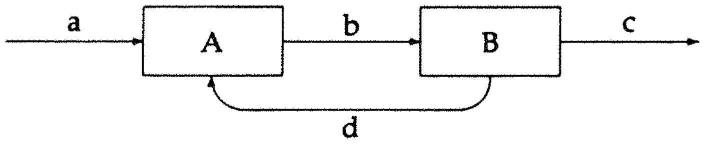


Similarly,

B = **unit requires** b **supplies** c, d

S is a structure built from A and B whilst hiding services b and d.

S = **structure** A, B **hides** b,d



4.2.2 IKBS Requirements

The overall requirements of a suitable IKBS for this problem can be clarified, using the above notation, as follows:

> In its usual mode of operation the IKBS will be given some partial knowledge about a group of systems. That is, it will know of a set of modules, a set of structures and perhaps some complete systems (i.e. complete knowledge about the systems). Given this knowledge, the first thing it will do is to infer the total information about the systems. During this process it is likely to come across inconsistencies (often typing errors). This is where integrity checking will help to remove errors from the input data. Having arrived at a set of completely defined systems it is ready to receive a specification for a desired new system. It must then identify suitable combinations of systems that meet that specification.

4.2.3 Rules of Composition

When components are composed to form a structure the resulting structure's services are governed by the *rules* or *semantics of composition*. These may be stated as:

1. The supplied services of the structure must equal the union of all the services supplied by its constituent components, minus any hidden services.
2. The required services of the structure must equal the union of all the services required by its constituent components, minus any services supplied by those constituent components.

As a corollary to the above semantics, it should be noted that this means that required services may be used up in some sense — they are *consumable* — whereas supplied services are not. Supplied services can be used over and over again and only appear consumed by being explicitly hidden.

4.2.4 Data Dependencies

By now it should be apparent to the reader that this problem has several interdependent pieces of data. Given a subset of the data it should be possible to infer the others. The following identifies those combinations:

- Given the structure of a system, i.e. its component systems and the services hidden, it is possible to infer the required and supplied services of the system using the *rules of composition*. These rules assume that the component systems are fully described in the knowledge base. If they are not fully described, they can be completed by using this or the other rules until the problem *bottoms out* at the primitive modules.
- Given the components of a system and the supplied and required services, it is possible to infer the hidden services.
- Given the supplied, required and hidden services, plus a database of fully described systems, it is possible to infer several structures that would produce those services.

- Given the supplied and required services (the specification of a desired system), plus a database of fully described systems, it is possible to infer several structures that would produce those services. (The hidden services may take any value.)

Due to these various dependencies, it is important to identify exactly how the IKBS will be required to function. A Prolog implementation did go some way to providing a more versatile IKBS, capable of carrying out all of these inferences (see section 4.8).

4.3 Problem Solving

To make the implementation easier the problem can be simplified by removing the *hidden* services. The following is a textual summary of the knowledge that must be represented for this problem solving exercise. This summary is the result of an analysis of a simple example on paper plus the experience of the first attempts at implementation.

4.3.1 Textual Summary of Simplified Problem

Given a set of *modules* with specified *supplied* and *required* services. Given a desired specification of a *system* in terms of the services it *supplies* and *requires*. The IKBS should find possible *structures* (*components* only, no *hidden* services) built from the primitive modules, such that:

- The *rules of composition* are satisfied. These are the same as in section 4.2.3, except for a change to rule 1.

  1. The supplied services of the structure must equal the union of all the services supplied by its constituent components.

- The final solution should meet the constraints of the desired specification, in this case a simple equality of the inferred value of the *supplied* and *required* services with the specified values.

The following knowledge must be represented:

1 If `m` is a module, it supplies and requires those services with which it has been declared.

2 If `m` is a structure, the supplied and required services are a function of the components of the structure, as governed by the *rules of composition*.

3 If `c` is a set of components that may form a solution, and the supplied and required services inferred using the *rules of composition* are equal (set-equal) to those of the desired system, then `c` is a solution to the problem.

If this is not the case `c` can be used as the basis for extending the search for a solution set of components.

The above is not enough knowledge to solve the problem: it does not say how to generate new sets of components that might be solutions, nor does it say the best way to search those possible solution sets.

One way to go about solving the problem is to start with an empty set, and add components to it in an attempt to produce the desired system. The possible components are checked to see if they meet the specification of the desired system; if not another component is selected and added to the possible components set. This continues until a solution is found. Once one solution is found, the user will either want the IKBS to stop searching for solutions or continue until any number of solutions are found.

So the problem solving involves a search through all the possible combinations of components the IKBS knows about. The question is how to guide and thus speed this search. The search can be helped by restricting the components that are allowed to be used in the set of possible components:

4 A component cannot be added to the set of possible components if it *supplies* a service that is in the desired system's *required* services. This can be seen from the *rules of composition*.

5 A component cannot be chosen if it is already part of the set of possible components. (If the SA wants several instances of the same component, he must name them individually.)

There are several pieces of derived information that will be useful in guiding the search. These should be calculated for each set of possible components and, together with the set of possible components, can be thought of as a *state* in *state space*. (The problem is now taking on the standard problem solving strategy of state-based search.)

6 The derived information (also called state-variables) for each state is:

```
TOTAL-SPEC_REQUIRES = inferred requires  - specified requires
TOTAL-SPEC_SUPPLIES = inferred supplies  - specified supplies
SPEC-TOTAL_REQUIRES = specified requires - inferred requires
SPEC-TOTAL_SUPPLIES = specified supplies - inferred supplies
```

which gives us

```
NEED_TO_REQUIRE     = SPEC-TOTAL_REQUIRES
NEED_TO_SUPPLY      = SPEC-TOTAL_SUPPLIES ∪ TOTAL-SPEC_REQUIRES
```

(N.B. This is set subtraction.)

It is now possible to describe three heuristics to help guide the search:

Heuristic 1: When selecting a system to add to the possible components, first try to choose one to reduce the NEED_TO_REQUIRE; if this is not possible, then try to choose one to reduce the NEED_TO_SUPPLY. Finally, if no system proves to be eligible for addition to the set of possible components, give up trying to extend this partial solution.

Heuristic 2: A system is eligible for inclusion in the set of possible components, if it is not one of those forbidden by 4 and 5 above and one or more of the services required by the system are in the NEED_TO_REQUIRE set.

Heuristic 3: A system is eligible for inclusion in the set of possible components, if it is not one of those forbidden by 4 and 5 above and one or more of the services supplied by the system are in the `NEED_TO_SUPPLY` set.

Corollary to Heuristic 1: If no system is eligible by Heuristic 2, and the `NEED_TO_REQUIRE` is not an empty set, then no solution will ever be found.

4.3.2 The Complete Problem

This problem is similar to the above except for the following changes:

- The complete *rules of composition*, as given in section 4.2.3, are satisfied.
- The final solution should meet the constraints of the desired specification, in this case a simple equality of the inferred value of the *supplied* and *required* services with the specified values. However, there is an extra degree of freedom due to the possibility of *hiding* any superfluous supplied services. Thus, if the inferred required services is set equal to the desired required services, and the inferred supplied services is a super set of the desired supplied services, the hidden services is the set difference between the inferred and desired supplied services, namely the `TOTAL-SPEC_SUPPLIES` set.

4.4 The Experiments

Henderson and Warboys' approach to the system design problem is to provide a language that allows the SA to create and explore new designs. In particular it hinges upon a constructive description of the new system from previous components, both derived and primitive. The implementations presented here started with this approach. A tool was developed which allowed the SA to construct new components and check their integrity. This meant that the *rules of composition* were represented explicitly within the tool; these were checked only at the user's request. If any inconsistency occurred, the tool helped the SA locate the erroneous description of a component.

Having implemented a tool to assist with maintaining integrity during component construction, the next step was to attempt to capture some of the knowledge pertinent to solving the SA's problem — "How to design components to meet a specification". The following is a history of the work. This thesis will be concentrating on the implementations in KEE.

1. The first step was to analyse "How to solve the problem" on paper. A simplified version of the problem was isolated, i.e. without the hidden services. The key input data and any derived data needed to assist in the problem solving were noted. An example was worked through on paper and key decisions noted. A state-based strategy seemed appropriate to the problem. Finding a solution became the process of searching the state space, being guided by a few heuristics. The results of this analysis have already been described in section 4.3.
2. A tool was implemented to check the integrity of the data which had been entered into the knowledge base by the SA. Any partially described systems were completed.

3. Having discovered the sketch of an algorithm for solving the problem (step 1 above), it was implemented in the LISP which underlies the KEE tool. It was possible to use the object oriented representation (frames), provided by KEE, to implement the data structures whilst the search algorithm was written in LISP.

4. Then an attempt was made to capture the same knowledge in KEE, making use of its many functional capabilities. That is a mixture of: the production rule system; message-passing and inheritance of object oriented programming; and the worlds mechanism.

5. The above approach proved to be complex and difficult to control, so an attempt was made in Prolog. This followed a *generate and test* strategy and proved succinct but lacked a simple way of introducing the heuristics.

6. Finally, Z was used to specify the problem solving knowledge. It was used to clearly define the state-variables and the state transitions. The specification was then implemented in KEE using a restricted set of KEE's functionality i.e. the object oriented paradigm for data description and any actions to be initiated by the user; the world mechanism to represent the states of the problem space; and the rule system (forward chaining only) for the state transitions and heuristics.

4.5 Integrity Checking Tool

The first experiment carried out was to build a tool that helped the user check the integrity (i.e. internal consistency) of the data to be used by the problem solving IKBS.

One of the first difficulties encountered was KEE's lack of a complete implementation of sets — the `set.of` operator. As a result it was necessary to use the `list.of` operator and write LISP functions to manipulate the lists as sets e.g. `my-union`, see appendix E.3.2.

In order to carry out the integrity checking of the components constructed with this tool, it was necessary to distinguish the declared supplied and required services from those implied by the tool. This resulted in a proliferation of slots e.g. `supplies` and `supplies.imp`. Attempts were made at using just the one slot, e.g. `supplies`, and writing different integrity rules. However, this was not successful, it being much simpler to keep the two kinds of knowledge distinct.

The tool also had the *rules of composition* represented twice: once in the `ACTION.INT` rules and again in the object oriented methods. Both were needed, the action rules for operation Mode 2 and the object oriented methods for Mode 3 (discussed below). Ideally the *rules of composition* should have been represented once and made use of by both the rules and methods. This was not possible since they were so intertwined with the paradigm of their use.

In practise it proved much easier to implement the semantics in an object oriented fashion. Implementation in the Rulesystem was prone to mistakes and misunderstanding of how the Rulesystem worked. The object oriented paradigm was much more easily understood and thus the solution easier to implement. An example of the Rulesystem's complexity was the radical change in the inferences made by changing a global flag affecting the Rulesystem's behaviour (this is discussed further in section 4.7); it was found difficult to keep track of exactly how the system would behave.

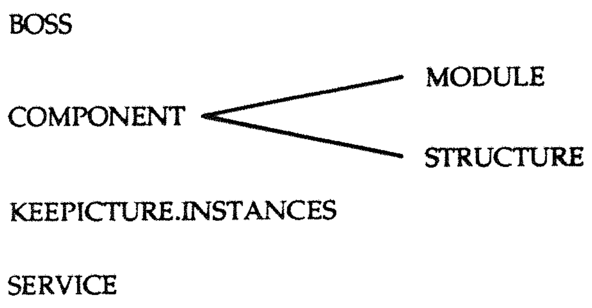


Figure 4.1: The Class Hierarchy.

It was possible to apply the integrity checking capabilities of this tool to any knowledge base, as long as the structure of the classes (i.e. the hierarchy and slots) was present — it was also possible to extend the structure if necessary. The integrity checker was thus independent of any problem solving capabilities given to the IKBS.

This tool was successful because it was still reasonably small, control being maintained by several devices:

- Processing and inference were broken down into small units.
- Those units were initiated by the user. This was done by the use of active images and methods (i.e. *buttons*).
- Small rule classes were used to segregate the rules into manageable groups. This prevented them interfering with each other in unexpected ways.
- The reasoning represented by each rule class was very *small/simple*.

The following sections discuss the details of this implementation; these may be skipped by moving to section 4.6.

4.5.1 Class Hierarchy

Figure 4.1 is a pictorial representation of the *class hierarchy* that was chosen for this tool. The objects the tool was to manipulate were inserted as children (or instances) of the SERVICE, MODULE and STRUCTURE classes. The BOSS object had several *methods* which helped control the tool. The KEEPICTURE.INSTANCES object was created automatically by KEE when *active images* were attached to the slots of classes or instances. The STRUCTURE class was used to contain instances of both *structures* and *systems*. See section 4.2.1 for clarification of notation.

The following is a brief summary of the slots that were in this hierarchy of classes.

BOSS Below is one of several *methods* that belonged to the BOSS object.

Slot	**Valueclass**	**Value**
rm_requires.imp!	Method	(lambda (self) (declare (ignore self) (loop.with.bindings (?c is in class component) do (unitmsg ?c 'rm_requires.imp!))))

This method, when activated by the user through the KEE interface, sent a message to all *objects* in the COMPONENT class to do the method in their own rm_requires.imp! method slot. In this case, all the implied required services of those objects would have been deleted. Methods like this one allowed the user to interact with the tool and carry out some programmed function.

COMPONENT This had several *member* slots; these were inherited by its *descendents* in the hierarchy.

Slot	Valueclass	Value
describe.m	Method	(lambda (self) (format *standard-output* "~& Unit = ~S ~% Type = ~S ~% Requires = ~S ~% Supplies = ~S" self (unit.parents self 'member) (get.values self 'requires) (get.values self 'supplies)))
requires	(list.of SERVICE)	nil
requires.imp	(list.of SERVICE)	nil
rm_requires.imp!	Method	(lambda (self) (remove.all.values self 'requires.imp))
rm_supplies.imp!	Method	(lambda (self) (remove.all.values self 'supplies.imp))
supplies	(list.of SERVICE)	nil
supplies.imp	(list.of SERVICE)	nil

The describe.m method asked the object to describe itself, giving details of the required and supplied services of its parent class. This method did *not* capture the semantics of composition for components, it just asked the component to display its slot values. Note the need to understand LISP.

There were also methods to help the user experiment/manage the data in the tool: such as those for the removal of the inferred supplied and required services. These had a "!" suffix.

There were slots for the supplied and required services, supplies and requires. Their values were restricted to lists of objects from the class SERVICE. This should have been sets of objects but unfortunately KEE did not have a complete implementation of sets.

It is important to note the introduction of slots for the inferred supplied and required services, namely supplies.imp and requires.imp respectively: these were critical in the integrity checking process.

MODULE This inherited all of the slots from the COMPONENT class. It also had the following:

```
Slot          Valueclass   Value

describe?     Method       (lambda (self)
                             (list 'module
                               (unitmsg self 'requires?)
                               (unitmsg self 'supplies?)))

requires?     Method       (lambda (self)
                               (get.value self 'requires))

supplies?     Method       (lambda (self)
                               (get.value self 'supplies))
```

Alfa had object oriented semantics that allowed the SA to construct systems and discover the services they supplied and required. These captured the *rules of composition* and were implemented as the methods with the "?" suffix. In this case, the `requires?/supplies?` methods captured the semantics of the required/supplied services for a *module*. These were trivial in this case but more complex for *structures*.

STRUCTURE Once again this inherited the slots from the COMPONENT class. The following were also added:

```
Slot          Valueclass        Value

components    (list.of          nil
                 COMPONENT)

describe.m    Method            (after
                                 (format *standard-output*
                                  "~& Components = ~S ~% Hidden = ~S"
                                  (get.values self 'components)
                                  (get.values self 'hidden)))

describe?     Method            (lambda (self)
                                 (list 'structure
                                  (mapcar
                                   #'(lambda (c)(unitmsg self 'describe?))
                                   (get.values self 'components))
                                  (get.values self 'hidden)))

hidden        (list.of          nil
                 SERVICE)

requires?     Method            (lambda (self)
                                 (set-difference
                                  (my-union
                                   (mapcar
                                    #'(lambda (c)(unitmsg c 'requires?))
                                    (get.value self 'components)))
                                   (my-union
                                    (mapcar
                                     #'(lambda (c)(unitmsg c 'supplies?))
                                     (get.value self 'components)))))

supplies?     Method            (lambda (self)
                                 (set-difference
                                  (my-union
                                   (mapcar
                                    #'(lambda (c)(unitmsg c 'supplies?))
                                    (get.value self 'components)))
                                   (get.value self 'hidden)))
```

The SA was able to construct systems (in this case instances of the class STRUCTURE) and ask the instance to `describe?` itself, i.e. flatten the structure to primitive modules. The SA could also ask which services the structure supplied and required using the `supplies?` and `requires?` methods.

4.5.2 The Rule Classes

Figure 4.2 shows the rule classes and instances; solid lines indicate the *subclass* relationship between *units*, the dashed lines indicate the *instance-of* relationship. The rules and methods resulted in three modes of operation:

Mode 1: Partial Knowledge & Object Oriented Semantics The SA entered the primitive modules and several structures (components and hidden services only) into

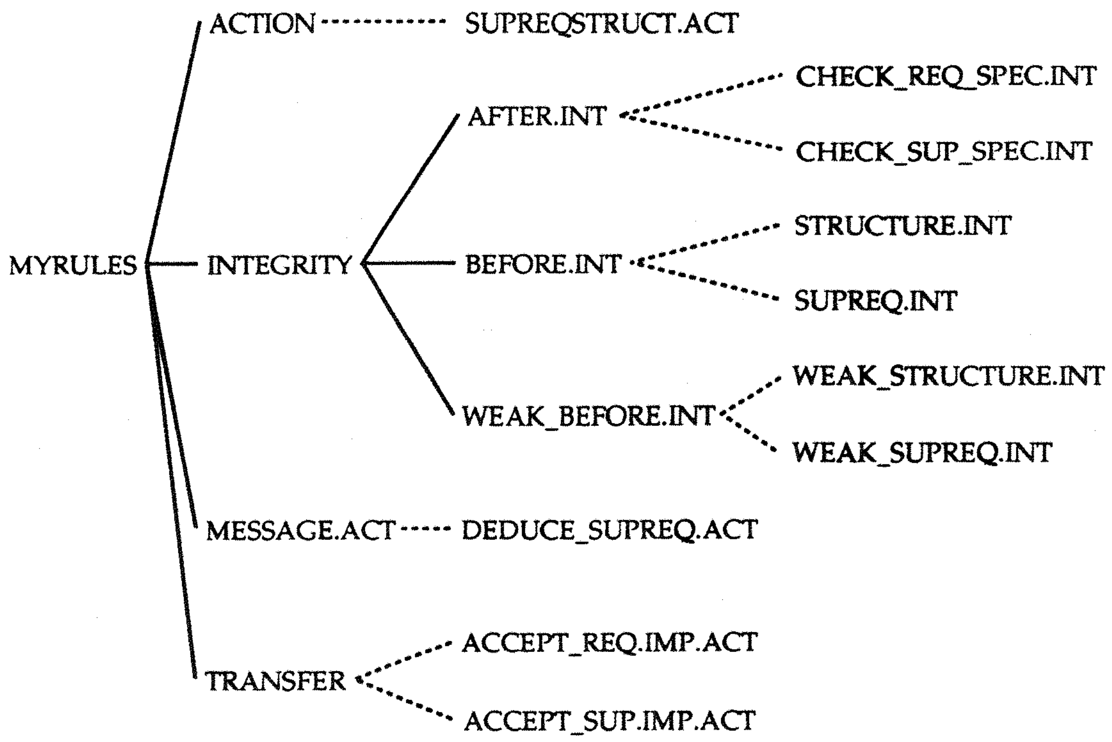


Figure 4.2: The Rule Classes.

the knowledge base (KB). Recall that a structure was only partial knowledge about a system.

The SA then interrogated the KB using the object oriented semantics as outlined above. Thus he discovered if the constructed structures did in fact supply and require those services he had hoped. If not, he could change the structures.

Mode 2: Complete Knowledge & Integrity Checking As well as the knowledge above, the SA entered the complete description of the systems, i.e. a structure plus the *requires* and *supplies*. He then checked the integrity of the data using a sequence of actions. These actions took the form of methods associated with the MYRULES class. They were triggered by *active images* attached to the MYRULES class. The methods initiated FW chaining with a particular rule class, e.g. the BEFORE.INT rule class. The following was a typical example of these methods:

```
(fire.before                                      ;; Slot
    (value
        ((lambda (self)                           ;; Method
            (declare (ignore self))
            (assert nil 'before.int)))))
```

The final line told the Rulesystem to try to fire all the rules in the BEFORE.INT rule class.

The sequence of actions was as follows:

Fire Before Integrity Rules These were collected as the BEFORE.INT rule class. These checked that all the instances of the STRUCTURE and MODULE classes in the KB had their appropriate slots filled. Thus all COMPONENTS should have had their supplies and requires slots filled and STRUCTURES should also have had their components and hidden services filled. If these rules found any empty slots, they displayed a message indicating which slot in which instance was to blame. This allowed the SA to correct the mistake.

An example of this kind of rule was STRUCTURE.INT:

```
(structure.int
  (if (?s is in class structure)
      (or (cant.find (find (the components of ?s is ?hp)))
          (cant.find (find (the hidden of ?s is ?hs))))
      then
      (lisp
        (progn
          (format *standard-output*
            "~&Structure.int rule finds bad integrity. ~%")
          (format *standard-output*
            "This is due to a components or hidden ~%")
          (format *standard-output*
            "slot of a structure ~S being empty. ~% ~%"
            ?s)))))
```

This said:

```
If
  ?s is in the structure class and
  either there is no value in the components slot of ?s or
```

```
          there is no value in the hidden services slot of ?s,
     then
       Carry out a Lisp procedure that prints out "information"
       describing the fault and where it is found.
```

Fire Action Rules These were collected in the rule class `ACTION.INT`. This action set off a FW chaining process that calculated the supplied and required services that were implied by the declared structure. This rule captured the *semantics of composition* for components.

```
(supreqstruct.act
  (if (?s is in class structure)
      (the components of ?s is ?l)
      (the hidden of ?s is ?hs-l)
      (?supplies.f =
           (lisp
             (my-union
               (mapcar
                 #'(lambda (component)
                     (get.value component 'supplies))
                 ?l))))
      (?requires.f =
           (lisp
             (my-union
               (mapcar
                 #'(lambda (component)
                     (get.value component 'requires))
                 ?l))))
      then
      (change.to
           (the supplies.imp of ?s is
             (set-difference
               ?supplies.f ?hs-l :test #'equal)))
      (change.to
           (the requires.imp of ?s is
             (set-difference
               ?requires.f ?supplies.f :test #'equal)))))
```

The important points to note here were:

- The use of second slots for the implied supplied and required services.
- How the rules carried out a calculation using LISP, e.g. the value of the `supplies.imp` slot.
- How these rules assumed that the required and supplied services of the components of a structure had been declared. The rules used the declared values of these slots (`requires` and `supplies`), from the nested components, to calculate the implied values (`requires.imp` and `supplies.imp`) for the structure in question. Thus any errors in the input knowledge would have caused a discrepancy between the declared and implied values.
- The use of the `change.to` operator — this caused a slot value to be changed rather than a new value added to the old one.

Fire After Rules These were collected in the rule class `AFTER.INT`; they checked that the declared and implied values of the `supplies` and `requires` slots were equal, raising an error otherwise. They identified the instance that contained the discrepancy so that the user could correct the mistake.

Mode 3: Partial Knowledge, Completion and Integrity Checking The SA entered a mixture of partially and completely declared structures. He then went through a similar sequence of actions to Mode 2.

- **Fire Weakened Before Integrity Rules** The integrity rules of Mode 2 were weakened so that structures no longer had to have their supplied and required services declared. These were collected under the `WEAK_BEFORE.INT` rule class.
- **Calculating the Implied Services** The action rules of Mode 2 would no longer work, since not every supplies and requires slot of a component would have a value. A new rule was created in the `MESSAGE.ACT` rule class. This found the value of the implied supplies and requires using the object oriented semantics of Mode 1. The implied values were no longer a direct consequence of the declared values of the next nested layer in the hierarchy, they now resulted from the semantics of composition and the complete structure of the component (which may have contained nested structures). A message was sent to every instance (component) to calculate its supplied and required services, the returned values were then placed in the appropriate implied slots.
- **After Integrity Check** These rules were also the same as Mode 2; they located any discrepancies as before. These identified those components that had implied values for the supplied and required services but no declared values.
- **Completing the Knowledge** The SA was then in a position to accept the implied values of the supplied and required services. He did this via another active image which fired the `TRANSFER` rule class. The following was one of the two rules.

```
(accept_sup.imp.act
  (if (?s is in class structure)
      (the supplies.imp of ?s is ?supi-1)
   then
      (change.to (the supplies of ?s is ?supi-1))))
```

The SA had now completed the declaration of the components in the KB and was assured of their integrity.

4.5.3 Worked Example 1

Worked example 1 was carried out using the tool working in Mode 3 (see appendix B.2.1). The tool successfully helped the user to identify inconsistencies in the knowledge base and arrive at a fully declared description of the components of interest. Those inconsistencies found were usually due to typing errors which often resulted from KEE's poor input interface. Future implementations would benefit from a simple parser or the ability to load details of instances from a structured file.

4.6 A Lisp Implementation

Before continuing with the implementation in KEE, it was decided to implement a problem-solver in Lisp, see appendix C. It was hoped that this would provide insight into the problem and a position from which to judge KEE. The problem was represented as a search through a state space. A breadth-first search strategy was implemented to find all the solutions. A state was represented as a list of the structure's name and the following state-variables (previously outlined in section 4.3):

- `TOTAL_SUPPLIES`: inferred supplied services
- `TOTAL_REQUIRES`: inferred required services
- `TOTAL-SPEC_SUPPLIES`
- `TOTAL-SPEC_REQUIRES`
- `SPEC-TOTAL_SUPPLIES`
- `SPEC-TOTAL_REQUIRES`
- `NEED_TO_SUPPLY`
- `NEED_TO_REQUIRE`

This problem-solver made use of KEE's frame-based representation for the data, i.e. classes and slots etc., thus it could only be used in conjunction with KEE. The problem solver was invoked from the *Lisp Listener* (an interactive LISP window) as follows:

```
(construct-structure 'Structure_name)
```

4.6.1 Worked Example 1

The LISP program was tested using worked example 1 (see appendix B.2.1). In order to test the program, a copy of a structure was made. The declared components and hidden services were deleted, and the IKBS asked to construct it. The output below shows the results for the `Unix` and `U_pc_ft` structures. The brackets after `"found solutions"` contain the suggested *components*. The second pair contain the *hidden* services.

```
> (construct-structure 'unix)

"found solution"
(USHELL UBOX)
"hidden-services"
(UAI)

"found solution"
(UBOX USHELL PC)
"hidden-services"
(PCAI UAI)

"found solution"
(PC USHELL UBOX PCKERMIT)
"hidden-services"
(VT100 PCAI UAI)

"found solution"
(PC KCLIENT KSERVER USHELL UBOX K)
"hidden-services"
```

```
(FILE_TRANSFER KPRO KCPRO PCAI KSPRO UAI)
NIL
>

> (construct-structure 'u_pc_ft)

"found solution"
(PC KCLIENT KSERVER UBOX TM K)
"hidden-services"
(KPRO RS232 UAI KCPRO PCAI KSPRO CONNECT VT100)

"found solution"
(PC KCLIENT KSERVER UBOX XOVER K)
"hidden-services"
(KPRO RS232 UAI KCPRO PCAI KSPRO CONNECT)
NIL
>
```

As the reader can see, the IKBS found 4 solutions to the `Unix` specification and 2 for the `U_pc_ft`. The SA could then choose which construction was the most appropriate. A more intelligent problem-solver might contain knowledge which would eliminate some of the possible solutions, e.g. by number of components, or cost.

4.6.2 Discussion

The LISP code captured enough knowledge about how to solve the problem to be successful. However, in order to change the search strategy or heuristics the user would have had to change the LISP code. This was because the knowledge had been described in a procedural fashion — "How to Solve The/This Problem" — and was thus difficult to modify or extend.

However, despite these drawbacks, this program took a relatively small time to implement compared to those using the KEE *Rulesystem* and *Worlds Mechanism*. This was probably due to the straightforward semantics of the LISP programming language; KEE's semantics were extremely complex and inter-dependent, due to the hybrid nature of the tool.

This program was restricted to considering combinations of primitive modules. It should really have allowed the nesting of previously constructed structures. This could have been included in future implementations.

4.7 First Attempt at an IKBS using KEE

Having implemented a solution in LISP, the next experiment attempted to capture the same problem-solving ability in KEE 3.1 on a Sun 4/110; making use of the *Rulesystem*, *Object Oriented* representation and the *Worlds* mechanism. The first attempt at this went through many different incarnations, none of which were particularly successful: the IKBS had become too large and complex.

It was difficult to control the reasoning process, in particular:

- It was not always clear which world was being searched for the premises of the rules, and similarly, in which the conclusions were being asserted.

- The search path was affected by several global variables:

Immediate Rule Application This affects BW chaining and can best be described with an example. Consider the two rules:

```
(if (p1)
    (p2)
then
    (c1) )

(if (p3)
    (p4)
then
    (p1))
```

If the initial goal is to prove `c1`, rule 1 will cause BW chaining to rule 2. If *immediate rule application* is *on*, the default, `p1` will be asserted as soon as `p3` and `p4` are found true. If it is *off*, `p1` will not be asserted until `p2` has been proved.

Node Redundancy Check This prevents a node in the search being created with the same goal stack as one that has been solved earlier. That is, it tries to prevent the goal being unnecessarily derived a second time. This may not always be appropriate.

Verify Rule Premise This is important at the stage when the FW chainer rule controller finally removes a rule to fire from the agenda. If this variable is *on*, it will check that the premises of the rule are still true. If it is *off* it will fire the rule even if some are now false.

- The order of premises in the rules affected the order of rule application in both FW and BW chaining: the rule controller searches for the premises/goals in the order they occur in the rules.

- KEE contains many bugs which resulted in strange rule execution, some of which were not even repeatable. These caused a great deal of time to be wasted. Those bugs and weaknesses include:

  - The `list.of` operator does not work properly.
  - Variable names can clash in different rules.
  - The `lisp` operator has to be entered explicitly, in rule premises, where it should occur automatically. This is used to surround pieces of LISP code.
  - Sometimes rules have to be manually re-parsed on editing or re-entering the knowledge base.
  - KEE does not detect misspelled variable or function names.

  It was thought that the experiments and observations made were still valid despite these bugs.

The following were used in an attempt to control the execution of the rules:

- A special `goal` slot, to represent different stages in the problem-solving.

- KEE's control operators and rule classes.

- Extra premises to prevent the wrong rules from firing.

There seemed to be a need to express the basic reasoning information in some form of simple rule whilst keeping higher level control knowledge separate. This was difficult to do in KEE: the control knowledge had to be expressed at the same level as the basic knowledge. This can be seen by looking at the rules below which included complex control operators, such as `find`, in amongst the premises and conclusions. This control knowledge should have been expressed separately. For example, GENERATE_NEXT_POSSIBLE_SOLUTION should have been expressed as follows (compare this with the complete rule shown in section 4.7.2):

```
(generate_next_possible_solution
  (if (the possible_solution of ?s is ?rest)
      (a possible_module of ?s is ?mm)
      then
      (the possible_solution of ?s is (list.of ?mm . ?rest))))
```

All the other information should have been expressed in another way. In a conventional programming language, the user is encouraged to use procedural abstraction to help break the problem down. It was an inability to do this, i.e. abstract different parts of the problem solving knowledge, that caused the difficulties with this knowledge base.

Despite a great deal of time and effort, this IKBS was never successful. The rules became more and more complex as attempts were made to control the reasoning process. The attempts were hindered by the complex semantics of the different programming paradigms offered by KEE. Moreover, the semantics of their interaction antagonized the situation. Despite KEE providing a complex rule tracing mechanism, it was still difficult to find where things were going wrong. All in all this attempt at representing, what was apparently a simple problem solving task, took too long and failed.

The following describes the culmination of these attempts: the reader is advised to read only a little since it is presented to give an impression of the complexity in which the programmer of KEE can become entangled. The text resumes in earnest at section 4.8.

4.7.1 Overview of Approach

The states of the LISP implementation were represented using KEE's *Worlds* mechanism. Slots were introduced, on the structure under construction, to represent the state-variables: these slots were to contain different values in different worlds. The principle representation of the reasoning process was the *Rulesystem*.

The search was started with a query for the components that could make up a specified structure, i.e. one with declared supplied and required services. This should have triggered a search through the state space, each intermediate state and solution state being represented by a KEE *World*.

The search process should have proceeded as follows:

- Take a state that is not a solution.
- Find a module suitable for inclusion in this partial solution.
- Create a new world and add the module to the partial solution.

- Infer the state-variables and assert them in the new world.
- Check if the state is a solution state, if so create a new world, marking it as a solution. Otherwise continue search.

4.7.2 Attempt at Implementation

1. The primitive modules and structures were described in a database, in a similar fashion to the previous experiment. The structure to be created had only its supplied and required services declared.

2. In an attempt to control the reasoning process, the rules were grouped together in rule classes. See figure 4.3.

3. The initial world, from which the search should start, was set up by firing the `initialise_state_m` method on the BOSS class. This carried out the following LISP code:

   ```
   (initialise_state_m
     (value
       ((lambda (self)
                (declare (ignore self))
                (retract '(the goal of ss is initialise_modules))
                (retract '(the goal of ss is initialise))
                (assert  '(the goal of ss is initialise)
                         'fw_initialise_rules))))
   ```

   This should have retracted two facts that might have remained from a previous session and asserted a fact to start FW chaining with the FW_INITIALISE_RULES. The goal slot, referred to in the method, had been introduced in an attempt to help control the rules. For simplicity, an explicit name was used for the structure under construction (SS).

4. The INITIALISE2 rule should then have fired due to its first premise. This should have created a new world and set up the state-variables. The goal should then have been changed to INITIALISE_MODULES.

   ```
   (initialise2
     (if (the goal of ?s is initialise)
         then
         in.new.world
         (change.to (the possible_solution of ?s is nil))
         (change.to (the poss_supplies.imp of ?s is nil))
         (change.to (the poss_requires.imp of ?s is nil))
         (change.to (the need_to_supply of ?s is
                                        (the supplies of ?s)))
         (change.to (the need_to_require of ?s is
                                        (the requires of ?s)))
         (change.to (the goal of ?s is initialise_modules)))))
   ```

5. The FW_INIT_MODULES2 rule should then have fired, asserting the implied supplied and required services for a module.

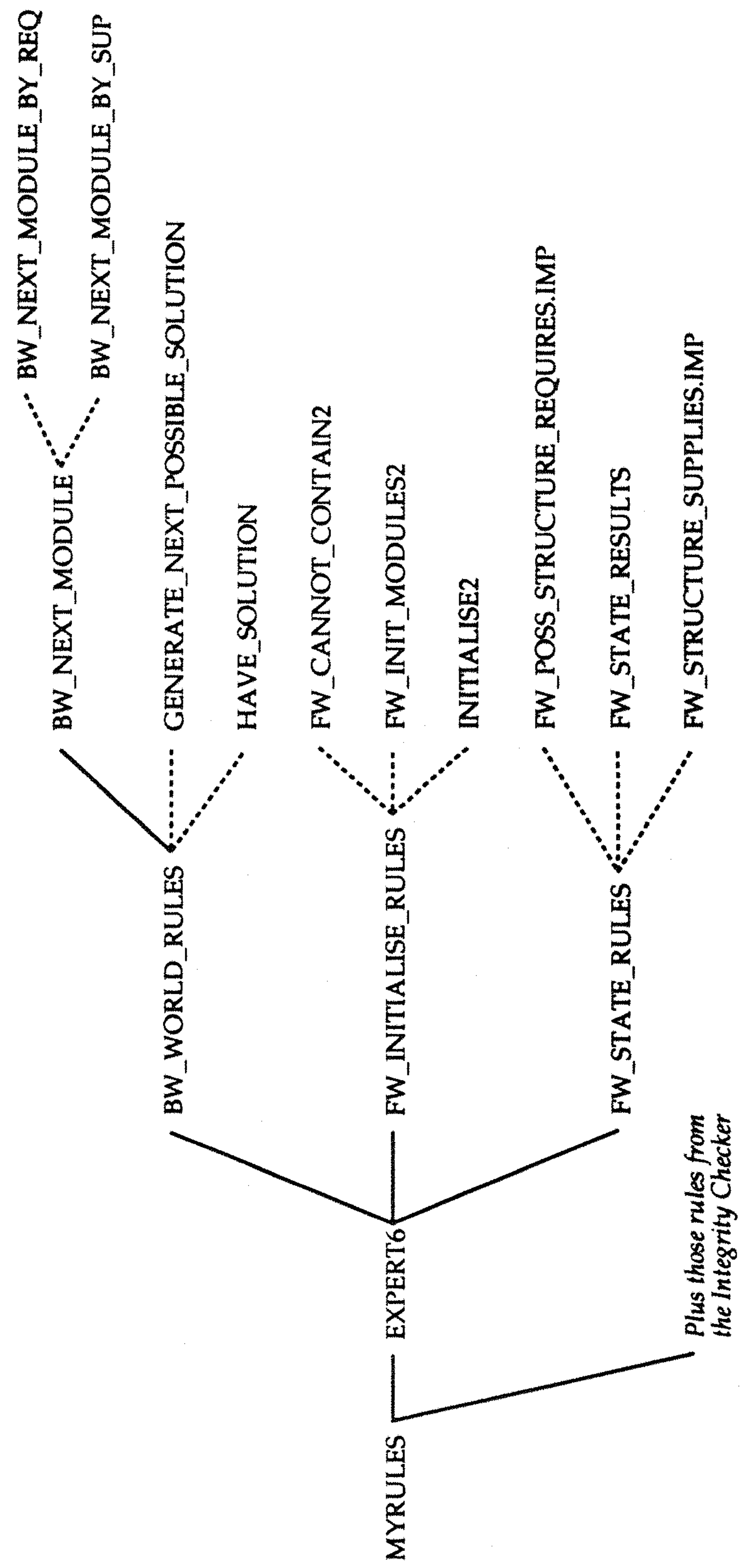


Figure 4.3: The Rule Classes.

```
(fw_init_modules2
  (if (the goal of ?s is initialise_modules)
      (?m is in class module)
      (the requires of ?m is ?req)
      (the supplies of ?m is ?sup)
      then
      (change.to (the requires.imp of ?m is ?req))
      (change.to (the supplies.imp of ?m is ?sup))))
```

6. This in turn should have triggered `FW_CANNOT_CONTAIN2` which should have attempted to discover all the modules that the structure could not have contained.

```
(fw_cannot_contain2
  (if (?s is in class structure)
      (the requires of ?s is ?req_l)
      (?m is in class module)
      (the supplies.imp of ?m is ?sup.imp)
      (lisp (not (null (intersection ?sup.imp ?req_l))))
      then
      (change.to (a cannot_contain of ?s is ?m))))
```

7. The BW chaining search for a solution should have been started by the following query:

```
(query '(the components of ss is ?what)
       'bw_world_rules           ;; rule class to use
       'initialise2-9)           ;; world to start search from
```

8. This should have matched the conclusion of the `HAVE_SOLUTION` rule in the `BW_WORLD_RULES` rule class.

```
(have_solution
  (if (the possible_solution of ?s is ?poss_sol)
      (the poss_supplies.imp of ?s is ?poss_sup.i)
      (the poss_requires.imp of ?s is ?poss_req.i)
      (the supplies of ?s is ?spec_s)
      (the requires of ?s is ?spec_r)
      (lisp (my-set-equal ?poss_sup.i ?spec_s))
      (lisp (my-set-equal ?poss_req.i ?spec_r))
      then
      in.new.world
      (the components of ?s is ?poss_sol)))
```

This rule should have added its premises as goals; it was hoped that the premises captured the relationships that should have existed, between the state-variables, for this to have been a solution. It had been assumed that the premises would need to exist in the same world and, if the conditions were met, that a new world would have been created to indicate the solution. This rule should have expressed the following:

> We have solution components `?poss_sol`, if a possible_solution `?poss_sol` has been found, such that its inferred supplied and required services (`?poss_supplies.imp` and `?poss_requires.imp`) are set-equal to the specified supplied and required services of the structure to construct, `?s`.

9. This was expected to first check if the initial world is a solution, fail, and cause a new world to be created using the GENERATE_NEXT_POSSIBLE_SOLUTION rule.

```
(generate_next_possible_solution
  (if (the possible_solution of ?s is ?rest)
      (find (a possible_module of ?s is ?mm)
                                        using bw_next_module)
      then
      in.new.world
      (delete (the possible_module of ?s is ?mm))
      (change.to (the possible_solution of ?s is
                                        (list.of ?mm . ?rest)))
      (change.to (the goal of ?s is fw_poss_supplies.imp)
                                        using fw_state_rules)))
```

The first premise of HAVE_SOLUTION should have caused BW chaining with GENERATE_NEXT_POSSIBLE_SOLUTION. This should have found a world with a possible-solution and tried to expand it by finding a suitable module:

```
(find (a possible_module of ?s is ?mm) using bw_next_module)
```

This should then have created a new world with that module added to the possible_solution set. The state-variables should then have been calculated using the FW_STATE_RULES.

10. The BW_NEXT_MODULE rule class should have tried to find a suitable module by using the required and then the supplied services still needed by the structure. Checking the value of the NEED_TO_REQUIRE slot (i.e. whether it was nil or not) should have provided the correct rule ordering. It had been assumed that both premises and conclusions would have been *searched for* and *asserted in* the current world.

```
(bw_next_module_by_req
 (if (the need_to_require of ?s is ?n_t_r)
     (lisp (not (null ?n_t_r)))
     (find (the possible_solution of ?s is ?poss_sol_r)
                                          using no.rules)
     (?mmm is in class module)
     (lisp (not (member (unit.name ?mmm) ?poss_sol_r)))
     (cant.find (a cannot_contain of ?s is ?mmm))
     (the requires.imp of ?mmm is ?req.imp)
     (lisp (not (null (intersection ?req.imp ?n_t_r))))
     then
     (add (a possible_module of ?s is ?mmm))))

(bw_next_module_by_sup
  (if (the need_to_require of ?s is nil)
      (cant.find (the need_to_supply of ?s is nil))
      (the need_to_supply of ?s is ?n_t_s)
      (find (the possible_solution of ?s is ?poss_sol_s)
                                          using no.rules)
      (?m is in class module)
      (lisp (not (member (unit.name ?m) ?poss_sol_s)))
      (cant.find (a cannot_contain of ?s is ?m))
      (the supplies.imp of ?m is ?sup.imp)
```

```
        (lisp (not (null (intersection ?sup.imp ?n_t_s))))
        then
        (add (a possible_module of ?s is ?m))))
```

These rules attempted to use KEE's control operators to restrict search. For example, the following should have prevented further BW chaining:

```
    (find (the possible_solution of ?s is ?poss_sol_s)
                                          using no.rules)
```

The `cant.find` operator, used above, does exactly as the name suggests: it tries to find a fact in the knowledge base, returning false if it is found and true otherwise.

11. So `GENERATE_NEXT_POSSIBLE_SOLUTION` should have extended the possible-solution by adding `?mm`. It should then have FW chained to change the state-variables:

```
    (change.to (the goal of ?s is fw_poss_supplies.imp)
               using fw_state_rules)
```

Note how the `goal` should have helped to control the reasoning process. The `FW_STATE_RULES` should have been ordered by the use of the `goal`, and should have proceeded as follows:

```
    (fw_structure_supplies.imp
      (if (the goal of ?s is fw_poss_supplies.imp)
          (the possible_solution of ?s is (list.of ?m . ?rest))
          (the poss_supplies.imp of ?s is ?poss_sup.i)
          (the supplies.imp of ?m is ?sup.i)
          then
          (change.to
               (the poss_supplies.imp of ?s is
                    (union ?sup.i ?poss_sup.i)) using no.rules)
          (change.to (the goal of ?s is fw_poss_requires.imp))))

    (fw_poss_structure_requires.imp
      (if (the goal of ?s is fw_poss_requires.imp)
          (the poss_supplies.imp of ?s is ?poss_sup.i)
          (the poss_requires.imp of ?s is ?poss_req.i)
          (the possible_solution of ?s is (list.of ?m . ?rest))
          (the requires.imp of ?m is ?req.i)
          then
          (change.to
               (the poss_requires.imp of ?s is
                    (set-difference
                         (union ?poss_req.i ?req.i) ?poss_sup.i))
               using no.rules)
          (change.to (the goal of ?s is results))))

    (fw_state_results
      (if (the goal of ?s is results)
          (the supplies of ?s is ?spec_s)
          (the requires of ?s is ?spec_r)
          (the poss_supplies.imp of ?s is ?total_s)
          (the poss_requires.imp of ?s is ?total_r)
```

```
                    (?s-t_r = (set-difference ?spec_r ?total_r))
                    (?s-t_s = (set-difference ?spec_s ?total_s))
                    (?t-s_r = (set-difference ?total_r ?spec_r))
                    (?t-s_s = (set-difference ?total_s ?spec_s))
                    then
                    (change.to (the total-spec_requires of ?s is ?t-s_r))
                    (change.to (the total-spec_supplies of ?s is ?t-s_s))
                    (change.to (the spec-total_requires of ?s is ?s-t_r))
                    (change.to (the spec-total_supplies of ?s is ?s-t_s))
                    (change.to (the need_to_require of ?s is ?s-t_r))
                    (change.to
                         (the need_to_supply of ?s is
                              (union ?s-t_s ?t-s_r)))))
```

These rules had been expected to find their premises in the current world, and assert their conclusions in the same.

12. Having generated the new possible-solution, HAVE_SOLUTION may or may not have succeeded. If it had failed, GENERATE_NEXT_POSSIBLE_SOLUTION should have continued to generate new states to test.

4.8 The Prolog Solution

Having failed to implement a successful IKBS in KEE, it was decided to attempt the same problem in Prolog. Once again the simple problem was attempted first and then extended to deal with hidden services; the complete solution is presented below. It was hoped that Prolog would allow an elegant specification of the problem. The problem solving strategy adopted was one of *generate and test*. A Prolog predicate would generate possible solutions, and further predicates applied to restrict the solution set.

4.8.1 The Program

The SA was expected to describe a set of primitive objects which the IKBS could use in its construction of the solution system. The SA posed the problem for the IKBS in the following form:

```
?- construct_system(Name,Set_of_requires,
                    Set_of_supplies,Components,Hidden).
```

This predicate was capable of receiving different combinations of variables and inferring the rest. Those combinations which worked were:

Name alone: It checked to see if the named system was known to the KB, if so it returned the appropriate details, otherwise it failed.

Required and Supplied Services: It generated solution pairs of components and hidden services.

Required, Supplied and Hidden Services: It generated solution components such that all three sets of services were consistent.

Name and either of the above two sets of data: It checked that the system was not already known before it constructed the above solutions.

Required, Supplied and Components: A system with these attributes was found if one was present in the KB. Otherwise, the appropriate value of the hidden services was calculated and a name generated.

4.8.2 Discussion

The declarative nature of Prolog facilitated the representation of the *semantics of composition* and the constraints between the supplied, required and hidden services in `is_solution_p`. The code was as follows:

```
% is_solution_p(?R,?S,+Set_of_systems,?Hidden_services):-

is_solution_p(R,S,Set_of_systems,Hidden_services):-
        setof(Supp,
              Sys^Rdum^Cdum^Hdum^(member(Sys,Set_of_systems),
                                  system(Sys,Rdum,Supp,Cdum,Hdum)),
              Set_of_supplies),
        union(Set_of_supplies,Stot),
        my_subtract(Stot,Hidden_services,S),
        setof(Req,
              Sys^Sdum^Cdum^Hdum^(member(Sys,Set_of_systems),
                                  system(Sys,Req,Sdum,Cdum,Hdum)),
              Set_of_requires),
        union(Set_of_requires,Rtot),
        subtract(Rtot,Stot,R).
```

Although the predicate was not totally reversible, that is independent of which data it was given and which it must deduce, Prolog did allow a certain degree of reversibility. Ideally, one would wish to express the constraints between the various variables, as in a mathematical equation, and the IKBS use these to discover the solution. Research is currently in progress on such *Constraint Programming Languages* [Lehler 1988]; these tend to be restricted to particular problem domains e.g. the solution of simultaneous equations.

The adoption of the *generate and test* strategy provided a powerful way of structuring the problem solving. This allowed the `construct_system` procedure, to be built up gradually; more predicates being added after the generation of a possible solution in order to enforce more constraints upon the solution.

The *generate and test* approach also had its short comings: it was not easy to include heuristics to guide the search. To do this would have required the expansion of the generator function. In this implementation, the generator included some constraints on the combinations of components generated. It did not contain any heuristic information, i.e. heuristics 1, 2 and 3, as described in section 4.3. Probably the best way to represent the heuristics, in Prolog, would be to build a shell (e.g. a rule interpreter).

The results to worked example 1 are shown in appendix D. In particular the first (and simplest) solution to the construction of a `unix` system was accomplished in about 5 seconds. The IKBS was still searching for other combinations after about 20 minutes.

The Prolog search mechanism was much more easily understood than that of KEE's Rulesystem3. This enabled me to develop the IKBS in a reasonable amount of time and with more confidence in the final results. All in all, the Prolog solution was more elegant, extendable, understandable and reliable than the previous attempts in KEE. It was not possible, however, to express the heuristics without building a *shell*.

It was decided to continue with the exploration of KEE, rather than build the Prolog shell.

4.9 An IKBS using Formal Specification and KEE

A second attempt was made to implement an IKBS in KEE. In order to keep the implementation manageable, Z was used to specify the problem and only a reduced set of KEE's functionality was used. Only the simplified Alfa problem was implemented, i.e. with no hidden services.

4.9.1 Z Specification Language

Z is a formal specification language used in the design and specification of computer programs. Its use has been explored by the Programming Research Group at Oxford University [Hayes 1987]. Z is built upon mathematical notation for logic and sets etc., plus the *Schema Calculus*. A *schema* provides a means of grouping together the declaration of variables and a predicate expressing constraints upon those variables.

Due to the problems experienced with the first attempt at implementing an IKBS in KEE, it was decided that as much of the problem as possible would be specified in Z. It was hoped that this would help manage the complexity of the problem and the implementation, resulting in a successful IKBS. It was also hoped to increase confidence in the results produced by the IKBS.

Appendix E contains the Z specification for the Alfa problem. Its relationship to the implementation is described below.

4.9.2 Implementation

A naive approach to solving the problem was implemented first (Z1 to Z11). This was then refined (Z12 to Z26). These are discussed in turn below.

In an attempt to keep the implementation simple, only a subset of KEE's functionality was used:

- Worlds were used to represent the possible states in state space; this provided a means of exploring different possible solutions.
- The search was implemented using the Rulesystem; this was restricted to forward chaining.

The two basic types of objects (see Z1), `MODULE` and `SERVICE`, were represented by two classes. See figure 4.4. The class `COMPONENT` was used to capture information pertinent to both `MODULE` and `SYSTEM`, i.e. the `supplies` and `requires` slots. These slots had *valueclass* restrictions imposed upon them; the values must be a `list.of SERVICE`. Ideally this would have been `set.of`, but KEE's implementation of sets was not complete.

The context of the problem was presented to the IKBS by declaring the `SERVICES` and `MODULES`. The latter having the appropriate values for their `supplies` and `requires` slots.

The *supplies* and *requires* functions, Z2 and Z3, were implemented in LISP — `supplies-f` and `requires-f` (see appendix E.3.1). The results of the functions were

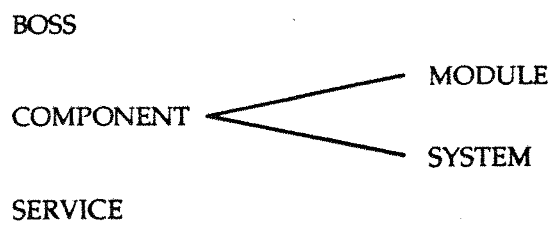


Figure 4.4: The Class Hierarchy.

type checked by the *valueclass* restriction of the slot in which they were to be recorded. N.B. The functions did *not* check that they had received a MODULE as argument.

The integrity check Z4 was not implemented. This could easily have been done with a method or rule. A query allowed the user to check that Z4 was not violated:

```
(query '( (?m is in class module) and
          (the requires of ?m is ?req) and
          (the supplies of ?m is ?sup) and
          (not (lisp (null (intersection ?req ?sup)))) ))
```

If this query found any module(s) with these properties, it was known that the constraint had been violated.

SYSTEM was introduced as a subclass of COMPONENT, and thus inherited the supplies and requires slots. Moreover, this class had a components slot. This latter slot had a *valueclass* restriction of a list.of MODULES. Once again, this should have been set.of. This slot varied from state to state as the search proceeded. The body of the *System* schema (Z5) was implemented as a predicate in LISP, see appendix E.3.1. This predicate was used to implement the *Solution* schema, see later.

The goal system was entered into the IKBS using the KEE assert function (Z6). This told the IKBS to start forward chaining using the set of rules that would control the search, namely EXPERT1. A special goal slot was introduced to the SYSTEM class to aid the control of these rules.

```
(assert '( (ss is in class system) and
           (the requires of ss is (a)) and
           (the supplies of ss is (b c d e)) and
           (the goal of ss is solve) )
           'expert1)
```

The states were represented using KEE's world mechanism (Z7). Only the goal and components slots of the system varied between states. The rules were written assuming that there was only one goal system (Z8).

Init0 (Z9) was implemented as an initialisation rule:

```
(if (the goal of ?s is solve)
    (find.any (?s is in class system))
 then in.new.world
    (add (the components of ?s is ())
    (change.to (the goal of ?s is search))))
```

Note how the goal was changed to search once initialisation had taken place.

Step0 (Z10) was implemented as follows:

```
(if (the components of ?s is ?c)
    (?m is in class module)
    (lisp (not (member ?m ?c)))
 then in.new.world
    (change.to (the components of ?s is (list.of ?m . ?c))) )
```

and *Solution* (Z11) as:

```
(if (the supplies of ?s is ?sup)
    (the components of ?s is ?c)
    (the requires of ?s is ?req)
    (lisp (system-p ?req ?sup ?c))
 then
    (change.to (the goal of ?s is solved)))
```

Note the use of a predicate, written in LISP, to capture the body of the *System* schema, see appendix E.3.1.

This search was started and left for several minutes. KEE was then halted. The solution systems were found using a `query` to KEE:

```
(query '((the goal of ?s is solved) and
         (the components of ?s is ?c)))
```

This resulted in an ungainly list of solution systems (KEE did not even pretty-print the result). E.g. worked example 2 (appendix B.2.2):

```
AA BB
AA BB DD
AA BB DD EE
AA BB DD EE SOL
AA BB DD SOL
etc.
```

All these systems required service A and supplied services B, C, D and E. As can be seen, this naive implementation found solution systems that contained subsets that were themselves solutions.

The Z specification was then refined. Each refinement was implemented in turn, and the effect on the search discovered and checked. This allowed the implementation in KEE to be kept under control and within comprehension. The refinements described by Z12 to Z19 were implemented as follows.

Z12, to prevent search continuing from a solution-state a premise was added to the *Step0* rule to form the *Step1* rule:

```
(if (the goal of ?s is search)
    (the components of ?s is ?c)
    (?m is in class module)
    (lisp (not (member ?m ?c)))
 then in.new.world
    (change.to (the components of ?s is (list.of ?m . ?c))) )
```

Thus the `goal` slot was used to restrict search: it was changed to `solved` in worlds (states) which were solutions.

However, this was not sufficient to completely implement Z12: KEE would still have found solution systems that were set equivalent. To prevent this, a complex condition had to be added to the *Step1* rule premises:

```
(?c_new = (cons ?m ?c))
(for
  (true.in.world
    ((the goal of ?s is solved) and
     (the components of ?s is ?c_old)) ?w)
  always
  (lisp (not (subsetp ?c_old ?c_new))))
```

This ensured that the new set of components was not a subset of the old. Problems were encountered in the representation of this condition. The initial attempt said:

```
(?c_new = (cons ?m ?c))
(cant.find
  (true.in.world
    ((the goal of ?s is solved) and
     (the components of ?s is ?c_old)) ?w)
  (lisp (not (subsetp ?c_old ?c_new))))
```

but it was not possible to wrap the `cant.find` operator around the `true.in.world` one.

In order to implement Z13 to Z15 an `eligible_modules` slot was introduced on the `SYSTEM` class. This had the valueclass restricted to a `list.of MODULE` (once again, this should have been `set.of`).

A function was implemented in LISP to construct the set of `eligible_modules` during the initialisation *Init1*, Z14.

```
(defun eligible_modules-f (comp requires)
  (letbindings ( (?em nil)
                 (?comp comp))
    (loop.with.bindings (?m is in class module)
        when (null (intersection (supplies-f ?m) requires))
             (lisp (not (member ?m ?comp)))
          do (setq ?em (cons ?m ?em)))
;; Return
   ?em))
```

The `eligible_modules` slot was maintained by the *Step2* rule, to meet the specification, Z15:

```
(if
   :
   (the eligible_modules of ?s is ?em)
   (?m is in class module)
   (lisp (member ?m ?em))
   (not (lisp (member ?m ?c)))
   :
   (?new_em = (lisp (set-difference ?em (list ?m))))
   (?c_new  = (lisp (cons ?m ?c)))
   :
   then
   in.new.world
   (change.to (the eligible_modules of ?s is ?new_em)
                                      using no.rules)
   (change.to (the components of ?s is ?c_new))))
```

The two heuristic specialisations to *Step2* were achieved by copying the *Step2* rule and adding extra premises. For example *Heuristic2*, Z16:

```
(lisp
  (not
    (null
      (intersection
          (requires-f ?m)
          (set-difference
              (set-difference
                  (the requires of ?s)
                  (union* (mapcar #'requires-f ?c)))
              (union* (mapcar #'supplies-f ?c)))))))
```

Z18, the ordering of the heuristics, was defined by using rule weights: when the rules are placed on the FC agenda, they can be ordered by rule weight. (The Rulesystem was discussed in more detail in section 3.8.3.) No attempt was made to implement the corollary to *Heuristic1*.

The complete rules are shown in appendix E.2.

4.9.3 Discussion

The use of Z with the restricted subset of KEE's functions, proved to be a successful way of developing the IKBS. This was helped by the adoption of a particular problem-solving strategy, namely a search through state space. Z was used within this strategy to help specify the following:

- The state-variables, that is those variables or slots that were to vary between states. N.B. It did not introduce the notion of a *goal* slot and said nothing about how KEE's control operators were to be used.
- The constraints existing between state-variables were also expressible in Z.
- The initialisation of a state.
- The transition/steps between states.
- The definition of a solution state.
- Heuristics to guide search. This was done by providing more than one state transition, and expressing the preference between these in English.

The KEE terms that corresponded to the above were:

- Slots took on the role of state-variables.
- A world was used to represent a state.
- Valueclass restrictions were used to express some of the typing constraints.
- Initialisation was carried out by a rule.
- The transitions/steps were also rules.
- LISP was used to implement functions.
- LISP was used to implement a predicate (`system-p` the body of the *System* schema).

- Identification of a solution state (or world) was done in a rule using the `system-p` predicate.

The experience gained from this experiment with Z suggests that the following are suitable steps for implementing an IKBS in KEE:

1. Specify the problem in Z.
2. Specify a simple implementation of a naive approach to solving the problem. It may take a couple of attempts to produce an elegant specification.
3. Refine the specification of the implementation into a more comprehensive solution.
4. Implement the naive problem-solver.
5. Test and correct using a simple example.
6. Then refine that implementation, in accordance with the refined specification. However, do not carry this out in one large step, do it piece by piece. Each extension of the problem-solver being tested — ensuring that it is in fact an improvement. Any problems that arise due to the extension can be corrected immediately.

This incremental approach to implementing the problem-solver kept the IKBS manageable and understandable. Because of the problem-solving strategy, the programmer could see each refinement as reducing the search space or imposing a tighter control on the order of that search. This constraining of the problem was a result of the forward chaining approach adopted: the programmer had to limit the growth of the search tree, reducing the potential paths that the search could take. This reduction was achieved by adding further constraints to the premises of the transition/step rules.

The main problems, encountered with KEE, were related to controlling the forward chaining rules (this is probably not specific to KEE). The means by which control was achieved were diverse and non-uniform. These included:

- The values of several global flags e.g. verify rule premise.
- How the rule system was told to search, breadth-first or depth-first, etc. In order to make the IKBS find single module solutions before multiple ones, it was necessary to ensure that verify rule premise was *on* and the initial assertion to the system indicated a breadth-first search:

```
(assert '( (ss is in class system) and
           (the requires of ss is (a)) and
           (the supplies of ss is (b c d e)) and
           (the goal of ss is solve) )
          'expert1
          nil
          ':agenda-controller 'fc.breadth.first)
```

- A goal slot was introduced, on the SYSTEM class, to allow extra control to be maintained over the rules. Thus the rules contained an extra premise just to help control them — "am I in the initialisation, search or check for a solution stage?". This is obviously a cumbersome method of representing the stages of a reasoning process.
- KEE's control operators were used to prevent assertions from causing superfluous forward chaining, e.g. the conclusions of the *step* rules. The `using.no.rules` operator, in combination with `change.to`, stopped KEE looking for further rules to fire due to this new data.
- The order in which the conclusions were asserted affected the order in which the rules were executed. KEE provides two ways to assert conclusions, *in order* or *together* (the latter removes the affect of the conclusions' order). To ensure that state changes associated with a *step* rule were carried out before further chaining occurred, one of two preventative measures were required:
  - the *rule type* was such that conclusions were asserted *together* before further forward chaining, or
  - the assertions were ordered, and all but the last were asserted `using.no.rules`. Thus only the last assertion would trigger further rules.
- There was no way of telling KEE that a particular fact, in a premise of a rule, was not one that should cause the rule to be *tickled*. This proved a problem with the *Init1* rule. It was necessary to use a LISP function instead of a slot value. The obvious way to express this was:

  ```
  (the requires of ?s is ?req)
  (?ms = (eligible_modules-f nil ?req))
  ```

  but the following had to be used instead:

  ```
  (?ms = (eligible_modules-f nil (requires-f ?s)))
  ```

All in all the control information was scattered between too many different mechanisms: goal slots; system flags; rule-ordering weights; the order of the rules in the database; the order of the premises and conclusions; control operators; rule type (*together, in.new.world, etc.*); where the rule controller looked for truth of premises and where the assertions were made. All this is far too complex.

4.10 Summary

The important points to note from this chapter are:

1. Although the design problem chosen (Alfa) was simple, it was difficult to successfully implement a problem-solver in the hybrid AI tool.
2. The integrity checking tool helped the user to check the internal consistency of the data presented to the problem-solver as explicit knowledge.

3. The object-oriented class hierarchy was an aid to the identification of abstractions and thus to the modelling of the problem domain.
4. Early identification of the problem-solving strategy made the problem-solver easier to implement.
5. The production rule system was complex and difficult to control; it was easy to write rules that apparently captured the domain knowledge which, in practise, failed to implement a problem-solver. These rules were not easily controlled or extended.
6. The interaction of the different representation formalisms was difficult to understand and predict.
7. The use of formal specification was a tremendous help in clarifying the problem and managing the hybrid AI tool's complexity.

5 Future Work and Conclusions

5.1 Future Work

The experiment using Z and a limited set of KEE's functionality appeared to be a promising approach to implementing IKBSs in hybrid AI tools. Further attempts at specifying both the Alfa problem and other problems, especially those that lend themselves to state-based search, should lead to the development of an elegant style of using Z for this combination of state-based search and hybrid AI tool.

Experience so far, suggests that it may be possible to identify a document template for the use of Z in specifying problems involving state-based search. The specification of the simplified Alfa problem approximately follows the structure below:

1. The problem is stated in English.
2. The problem is specified.
   (a) The data types are identified.
   (b) The attributes of the data types are identified and access functions defined.
   (c) Constraints upon the data types are expressed.
   (d) The problem is specified in a single schema, identifying the inputs, outputs and constraints between them.
3. The implementation is specified.
   (a) Inputs.
   (b) Outputs.
   (c) Identify state-variables: those that change with state.
   (d) Identify context-variables: those that remain constant with state change.
   (e) Initialisation of the state space.
   (f) Specify a simple transition in the state space: how to step from one state to the next.
   (g) Specify what constitutes a solution state.
4. The implementation is refined.

   The transition is refined by adding preconditions to prune the search-space. It may be necessary to introduce new state-variables to help guide search and thus reduce the search-space.
5. The heuristics are expressed in the following manner:
   (a) The single transition is refined into two or more different transitions.

(b) Any heuristic, expressing a preference for one transition over another, is stated in English.

(c) The conditions under which search terminates may be expressed as a theorem. This may be implemented later, as preconditions to the transitions, in order to stop search. (This was never completed for the experiment reported earlier.)

6. The Z is summarised.

Having completed the details of the above document, the programmer could implement the problem-solver using KEE's frames, rules and LISP. Hopefully it would be possible to identify a mapping from Z to KEE. If KEE proved not to supply all the representation structures required, it might be necessary to use one of the other hybrid AI tools or even build a specialised tool (in one of the hybrid AI tools, LISP or Prolog). This tool could be designed specifically for implementing solutions to problems amenable to the state-based search approach. It should then be easier to identify a suitable mapping to aid implementation.

It is expected that Z would help the knowledge engineer or programmer crystalise the details of the problem-solving strategy before implementation. The mapping, from the structured document and Z, to KEE (or the tool) should make the implementation easier. The programmer would also have more faith in the performance of the IKBS. He might also be able to prove certain properties, about the IKBS, using the Z.

5.2 Conclusions

The experiments reported here with KEE and the experience with Sowa's conceptual graphs, emphasise the need for the knowledge representation to have a well-defined semantics: a structure in the knowledge representation should clearly have a single meaning to the reader and writer of that structure. Similarly, the rules-of-inference must also be well-defined and not rely on ad hoc procedures.

Experience has shown that the process of conceptual analysis, although critical to the design of an IKBS, can consume an inordinate amount of time. The use of some form of class hierarchy, as found in KEE's frames, helps the programmer form abstractions of the real world. The abstractions hiding extraneous details thus allowing him to cope with some of the problem's complexity.

The experiments, presented here, have used two problem-solving strategies, namely *generate and test* and *state-based search*. It has been found that the IKBS is easier to implement when the programmer has a clear idea of the problem-solving strategy in mind. He is then better disposed to identify how the problem in question transfers into his problem-solving strategy.

KEE is a typical example of some of the hybrid AI tools currently available commercially and one with which I had had several months previous experience. Its hybrid nature, although appearing to give the benefit of many different programming paradigms, was in fact a hindrance to the successful implementation of an IKBS. The task of learning the semantics and peculiarities of each paradigm individually was difficult enough, but when the paradigms interacted with each other in such complicated fashions, it became too much for all but the most dedicated user ("guru") to fully comprehend. Future AI tools should take measures to keep the semantics of each paradigm as simple as possible, ensuring that the semantics of all the paradigms are unified in some fashion.

KEE's complexity was finally tamed by using a reduced set of its functionality and specifying as much of the problem as possible in Z. Future work in this area, as described above, should lead to a methodology that would help to control such large tools.

The work, presented here, suggests that care should be taken to select the correct AI tool for the application: KEE is one of several complex tools that, although powerful, may not result in a successful application unless programmed by a very experienced user; less complex AI tools may provide adequate computing power, to solve the particular problem, without the overheads associated with learning a complex programming environment. The majority of the problems with KEE stemmed from its production rule system, emphasising: the need for a better means of representing control knowledge; and that the production rules are by no means as extendable as software companies would suggest.

As IKBS systems become more widely used for complex tasks, they will undoubtedly be called upon to have a reliable performance. This work suggests that, as with any other kinds of computer software, care should be taken in the design, implementation and testing of the application. An increase in the use of formal specification in the IKBSs design and implementation should provide more confidence in their performance.

A Conceptual Graphs

A.1 Introduction

Knowledge representation is one of the most fundamental areas of research in Artificial Intelligence today. One appealing approach to this is to try and capture the associations between ideas. This led Quillian to form his theory of semantic networks [Quillian 1966]. Work started in the late 70s to give the concepts, in these networks, more meaning/structure. Minsky's *frames* can be thought of in this light [Minsky 1975]. This led to formalisms which Brachman has called "Structured Inheritance Networks" [Brachman & Schmolze 1985]. (The distinctive feature of these approaches, is that the description of the type's meaning is kept separate from the assertions made about individual entities of that type).

Sowa's Conceptual Graphs is one such formalism [Sowa 1984]. [1]

A.2 Conceptual Graph Theory

A.2.1 The Philosophy Behind the Formalism

In his book [Sowa 1984], Sowa presents psychological and philosophical arguments for his approach to knowledge representation. These arguments aside, he had another good reason for developing his notation: since Quillian's work there had been a proliferation of different graph based notations, with each researcher tackling a particular aspect of the knowledge representation (KR) problem. Sowa tried to unify the best features of these different individual formalisms to give a single notation in which to address those aspects of KR. He thus hoped to provide a framework for different areas of research, centered around a common notation for the communication of results.

The best way to get an overview of the formalism is to take a leaf out of Brachman's book and consider the knowledge representation language in two parts. The first is a descriptional part: this is how the *concept types* and *relation types* acquire their *meaning*. This meaning is gained via several definitional mechanisms in terms of other concept and relation types. The second is an assertional part: this is where assertions are made about particular *individual concepts* and their relationships to one another. For simplicity, it can be thought of as a collection of believed facts about *typed* individuals.

The next question is how to carry out inference? Sowa develops a system based upon Peirce's Existential Graphs [Roberts 1973]. The inference is carried out by manipulating the assertional graphs.

[1]The first publication about Conceptual Graphs was in Sowa 1976 although there is an unpublished paper Sowa 1968.

A.2.2 Assertional Part — Concepts, Relations and Logic

For simplicity, the linear notation for the graphs will be used rather than the 2 dimensional pictorial notation. This means that variables will have to be introduced if the graph is cyclic, whereas the pictorial notation would have shown the cycles directly.

A concept consists of two parts: the *typefield* and the *referent*. The typefield indicates the *concept type* of the individual. The referent indicates which particular individual the concept is referring to — there is a unique *individual marker* associated with each individual e.g. #007. Consider an example of an *individual concept*, "the block #1":

$$[BLOCKS\text{: } \#1]$$

This says that there is an individual #1, of type *BLOCKS*. In attempting to clarify the meaning of the graphs, Sowa developed an operator that would transform simple graphs into First Order Logic (FOL). The typefield becomes a predicate and the individual marker a constant. In this case:

$$blocks(\#1)$$

If one wanted to represent "a block", a *generic concept* would be used:

$$[BLOCKS\text{: } *]$$

where $*$ is called the *generic referent* (this is the default if no individual marker is present). This generic referent maps onto a variable in FOL:

$$\exists x\ blocks(x)$$

Note the implicit existential operator.

Relations show the relationships that hold between individual concepts. The relations are also 'typed'. They become two-placed predicates in FOL.

Consider some example graphs:

- "Block #1 is on a table"

$$[BLOCKS\text{: } \#1] \rightarrow (ON) \rightarrow [TABLES\text{: } *]$$

$$\exists x(blocks(\#1) \wedge tables(x) \wedge on(\#1, x))$$

- "Block #1 is on table #100"

$$[BLOCKS\text{: } \#1] \rightarrow (ON) \rightarrow [TABLES\text{: } \#100]$$

$$blocks(\#1) \wedge tables(\#100) \wedge on(\#1, \#100)$$

A.2.3 Descriptional Part — Concept and Relation Types

Sowa emphasises the meaning of concept types over relations, in both cases the 'programmer' or 'user' of the system must clearly understand that meaning.

A.2.3.1 Concept Types

To understand the meaning of a concept it is necessary to draw the distinction between the *intension* and *extension* of a concept type. The intension of a concept refers to the *essence* of that concept — how it is defined, what does it mean to be an individual of that type. The *extension* refers to the set of individuals that are of that type. For example, the intension of a block may be that "it has 6 faces, is coloured red, and has to be on top of another physical object". Whereas, in a particular domain there may only be two blocks under consideration, block #1 and block #2. Individuals #1 and #2 form the extension of the concept. From this, it is known that block #1 will have 6 faces, be coloured red and will be on top of something else.

When a domain is modelled the concept types must be given intensions; Sowa provides several ways for a concept to acquire substance from a collection of other concepts — each is a form of *abstraction*:

Definition A concept is defined in terms of another conceptual graph. This provides both *necessary* and *sufficient* conditions for a concept to be of this type. E.g.

> **type** *PEDESTRIAN* (*x*) **is**
> [*PERSON*: **x*] ← (*AGNT*) ← [*WALK*: *] → (*LOC*) → [*STREET*: *].

Necessary means, that if there is an individual of type *PEDESTRIAN*, then that individual *must* both be of type *PERSON* and be walking in a street. *Sufficient* means, that if there is an individual *PERSON* that is walking in a street, then that individual *must* be of type *PEDESTRIAN*.

Canonical Graph This is similar to the defining graph, except it only provides necessary and *not* sufficient conditions.

e.g. Canonical graph for *PEDESTRIAN*:

> [*PEDESTRIAN*: *] ← (*AGNT*) ← [*WALK*: *] → (*LOC*) → [*STREET*: *].

So, "if there is an individual of type *PEDESTRIAN*, that individual must be the agent of a *WALK* (or subtype thereof) and in a *STREET* (or subtype thereof)." *Subtype* is discussed below.

Schema These express a perspective on how a concept type *may* relate to others. A collection of schema for one concept forms a *schematic cluster*. For example, one schema for a *SCREW-DRIVER* will express its use 'to screw screws into things'. Another might express its use 'as a hammer'.

Prototypes These show the form of a typical individual. They specify defaults that are true of a typical case — they may not be true of any specific case.

Aggregation These are used to define a *composite individual* by specialising concepts in the definition graph of a type. A composite individual consists of other individual concepts. For example, consider the definition of *CIRCUS-ELEPHANT* and then the composite individual *CIRCUS-ELEPHANT:Jumbo*.

> **type** *CIRCUS-ELEPHANT* (*x*) **is**
> [*ELEPHANT*: **x*] ← (*AGNT*) ← [*PERFORM*] → (*LOC*) → [*CIRCUS*].

individual *CIRCUS-ELEPHANT(Jumbo)* **is**

[ELEPHANT:Jumbo] ← *(AGNT)* ← *[PERFORM:* {∗}] –
→ *(LOC)* → *[CIRCUS: Barnum & Bailey]*.

Another powerful device is to organise the types into a lattice using the *subtype* relation, <, (inverse is *supertype*). For example, *PEDESTRIAN* < *PERSON*, says that "the type *PEDESTRIAN* is a subtype of the type *PERSON*". From this it can be inferred that if an individual *conforms* to the type *PEDESTRIAN*, it also *conforms* to the type *PERSON*. It should be noted that this inference does *not* work in the other direction *and* that the relation is not a *subset* relation. (A type *T1* and a referent *R1* are said to conform if *R1* has appeared as a referent of *T1* or a subtype of *T1*.)

Two types may have common subtypes, in which case it is possible to define the *maximum common subtype* (denoted by ∩), e.g. *CAT* ∩ *PET* is *PET_CAT*. Similarly one can define the *minimum common supertype*, e.g. *CAT* ∪ *DOG* is *ANIMAL*. Care should be taken not to confuse these symbols with those of set-union and set-intersection.

For some pairs of concept types there is no obvious maximum common subtype or minimum common supertype, e.g. *CAT* ∩ *JUSTICE*, thus two special concept types are introduced. The *universal* type (top ⊤) which is a supertype of all other types and the *absurd* type (bottom ⊥) which is a subtype of all the other types. Now *CAT* ∩ *JUSTICE* is defined as the absurd type.

These mechanisms all contribute to the concept types intension. A 'user' may wish to query the system about these definitions, e.g "Where does a pedestrian walk?".

When a graph is loaded as an assertion, the system gains knowledge about the extension of concept types. It is this extension that allows the system to answer questions about the current state of the domain being modelled, e.g. "How many blocks are there?", "How many physical_objects are there?". Thus any implementation must take care of recording this information.

A.2.3.2 Relation Types

As indicated earlier, Sowa is a bit weak when it comes to describing the intension of a relation. Once again he allows relations to be defined in terms of other relations, these definitions again provide necessary and sufficient conditions, for example:

relation *AGNT(x,y)* **is**

[ACT: ∗x] ← *(LINK)* ← *[AGENT: ∗]* → *(LINK)* → *[ANIMATE: ∗y]*.

Sowa gives a list of relation types in an appendix [Sowa 1984]. For each type he gives the *maximal types* that may be linked to the arcs of the relation, an informal comment about the use of the relation and an example of the relation in a conceptual graph. For example,

agent. *(AGNT)* links *[ACT]* to *[ANIMATE]*, where the *[ANIMATE]* concept represents the actor of the action. Example: *Eve bit an apple.*

[PERSON: Eve] ← *(AGNT)* ← *[BITE: ∗]* → *(OBJ)* → *[APPLE: ∗]*.

This hand-waving description of a relation is one of the more blatant examples of why Sowa's theory of conceptual graphs is incomplete: the idea of a *relation* is intuitive and appealing but *not* sufficiently formalised to allow implementation.

Another piece of a relation's intension, is the number of individuals that can partake in the relationship. Sowa gives no way of providing this information. Brachman & Schmolze 1985, provide type restrictions, on a relationship, by what they call "Value Restrictions" and the limit on the number of individuals involved is provided by "Number Restrictions".

A.3 Conclusion

Sowa says that the mechanisms, described above, all contribute towards a concept type's intension. However, despite his discussion of model theory and inference based upon Peirce's graphs, he does not provide a unified theory of how these mechanisms should be used in an implementation.

B Problems

B.1 The Blocks World

The Blocks World is a classic *toy* problem that has received much attention throughout the history of AI. It is referred to as a *toy* problem because it is a simplified version of a problem that might occur in the real world. AI researchers hoped that they could use their understanding of the toy problems to help solve larger real problems. Terry Winograd was the first person to look at the Blocks World problem during his research into computer understanding of natural language [Winograd 1972]. The problem became so popular that it now appears in most standard AI text books, e.g. Charniak & McDermott 1985.

The Blocks World domain is one in which a robot arm moves blocks around a table top, see figure B.1. It is usual to view this as a *plan formation* problem: the computer system being responsible for producing the plan of actions which the robot arm must carry out.

The problem solver would typically be given explicit knowledge of:

- The blocks' colours, shape and size.
- The size of the table top.
- The initial location of the blocks *on* and *above* the table.
- The legal movements of the blocks e.g. the robot arm can only put a block on top of another block if neither have any other blocks upon them. (This knowledge captures the *physics* of the problem domain.)

A typical *goal*, which might be given to the problem solver, would be to:

Put block A on block B.

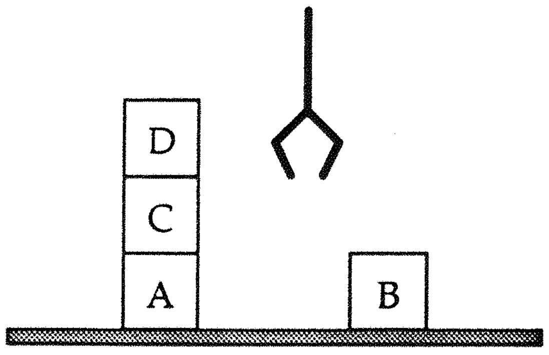


Figure B.1: The Blocks World.

Looking at figure B.1, we see that this is only possible if the problem solver first removes blocks D and C from above A. Thus a solution plan for this problem will contain *actions* that clear D and C from A, placing them somewhere on the table (other than on B!).

B.2 The Alfa Design Problem

B.2.1 Worked Example 1

The following is an example taken by kind permission from Henderson & Warboys 1989a.

Consider how we communicate with the Unix shell using a terminal. The components involved are a workstation *Ubox* and the shell itself *Ushell*. The workstation supplies the Unix application interface *uai* and a serial connection *rs232*, it requires to be connected to a terminal, a service we refer to as *connect*. The shell requires the Unix application interface and supplies the Unix commands *uc*.

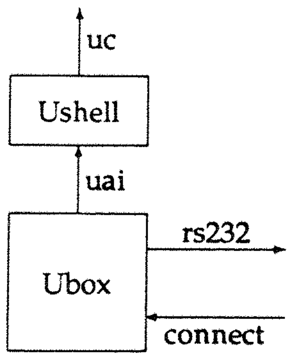


Fig B.2: Unix.

Unix	=	**structure** Ubox, Ushell **hides** uai
Ubox	=	**unit requires** connect **supplies** uai, rs232
Ushell	=	**unit requires** uai **supplies** uc

The terminal *TM*, when supplied with *rs232* provides a particular screen service *vt*100 and the *connect* required by *Unix*.

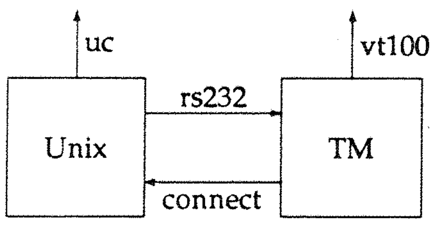


Fig B.3: Unix-TM.

Unix-TM	=	**structure** Unix, TM **hides** rs232, connect
TM	=	**unit requires** rs232 **supplies** connect, vt100

Kermit is a program which, when run on a PC, provides among other things VT100 emulation. We assume the PC provides an application interface *pcai* and the same combination of *rs232* and *connect* as a workstation. Kermit requires the *pcai* and also *connect* before it can supply *vt100*.

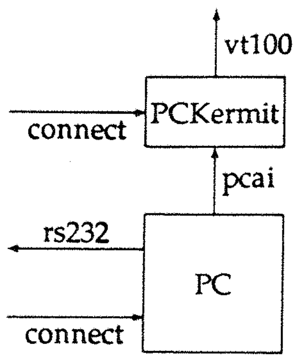


Fig B.4: pseudoTM.

pseudoTM	=	**structure** PC, PCKermit **hides** pcai
PC	=	**unit requires** connect **supplies** rs232, pcai
PCKermit	=	**unit requires** pcai, connect **supplies** vt100

To connect the *pseudoTM* to *unix* we must introduce a crossover *Xover*, which given the service *rs232* supplies *connect*.

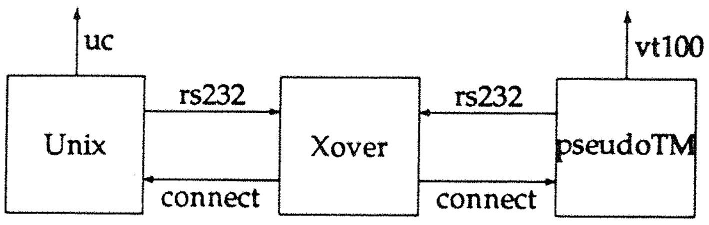


Fig B.5: Unix-pseudoTM.

Unix-pseudoTM	=	**structure** Unix, Xover, pseudoTM **hides** connect, rs232
Xover	=	**unit requires** rs232 **supplies** connect

We also use Kermit for file transfer. This is done by putting a Unix based version of Kermit into server mode, in communication with the PC based Kermit, which is in client mode. We describe this structure in two layers. First we connect the two machines.

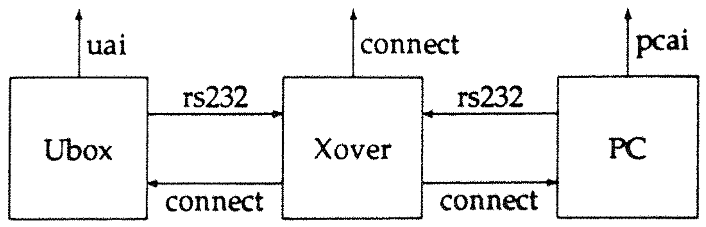


Fig B.6: layer1.

layer1 = **structure** Ubox, Xover, PC **hides** rs232

The second layer requires us to name some of the services provided by the Kermit file transfer protocol. We introduce a version of Kermit which runs on the Unix machine, call it *Kserver* and which provides the server part of the protocol which we call *Kspro*. In client mode, Kermit runs on the PC providing the client half of the protocol which we call *Kcpro*. We join these two protocol halves by an abstract component *K* which, given these two halves supplies the whole protocol *Kpro*.

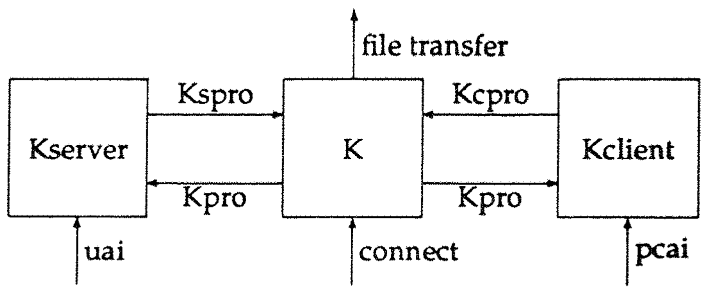


Fig B.7: layer2.

layer2	=	**structure** Kserver, K, Kclient **hides** Kspro, Kcpro, Kpro
Kserver	=	**unit requires** uai, Kpro **supplies** Kspro
K	=	**unit requires** connect, Kspro, Kcpro **supplies** Kpro, file transfer
Kclient	=	**unit requires** pcai, Kpro **supplies** Kcpro

Finally we assemble the two layers.

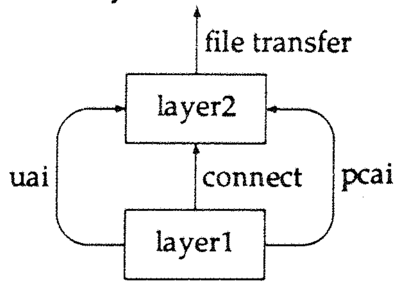


Fig B.8: U-PC-FT.

U-PC-FT = **structure** layer1, layer2 **hides** uai, pcai, connect

B.2.2 Worked Example 2

This was a simple test example containing 6 SERVICES, A, B, C, D, E and F, and 6 MODULES, AA, BB, CC, DD, EE and SOL.

AA	=	**unit requires** A, D **supplies** B
BB	=	**unit requires** B **supplies** C, D, E
CC	=	**unit supplies** A, B
DD	=	**unit requires** A **supplies** B
EE	=	**unit requires** B **supplies** C
SOL	=	**unit requires** A **supplies** B, C, D, E

The desired system should require service A and supply services B, C, D and E. The single module, SOL, was a solution to the problem. Other solution systems include a composition of AA and BB, and a composition of BB and DD.

C Lisp Problem Solver

```
 ;; Expert System
 ;;
 ;; 11/7/89
 ;;
 ;; This is the lisp solution to an expert system approach to
 ;; satisfying the specification of an alfa structure.
 ;; It will only work in conjunction with KEE.

 (in-package 'kee)

 (defun unit* (l)
   (cond ( (null l) nil)
         (t (cons (unit.name (car l)) (unit* (cdr l))))))

 (defun construct-structure (structure)
   (initialise structure)
   (let* ( (initial-state (list structure nil nil
                                  (get.value structure 'supplies)
                                  (get.value structure 'requires)) )
           (first-states-l (extend-search-of-states initial-state nil)) )
     (letbindings ( (?s structure)
                    (?components
;;                   (mapcar #'unit*
                             (find-solutions first-states-l nil)
;;                           )
                    ))
;;                  (assert '(the possible_components of ?s is ?components))
)))


(defun initialise (s)
  (assert nil 'module.imp.rules)
  (put.value s 'cannot-contain
             (unit* (cannot-contain-f s))))


;; A module that supplies a service that the specification of the
;; structure requires, cannot be in the construction of the structure.
;; Calculate this set, record it, and use it later to reduce search space.

(defun cannot-contain-f (structure)
  (let ( (requires-spec (get.value structure 'requires))
         (c-l           (unit.descendants 'module 'member)) )
    (cannot-contain-rec c-l requires-spec)))

(defun cannot-contain-rec (c-l requires-spec)
  (cond ( (null c-l) nil)
        ( t (append (cannot-contain-do (car c-l) requires-spec)
                    (cannot-contain-rec (cdr c-l) requires-spec)))))
```

```
(defun cannot-contain-do (c requires-spec)
  (let ( (c-req (get.value c 'supplies)) )
    (cond ( (not (null (intersection c-req requires-spec))) (list c))
          (t nil))))


;; Each structure will have a specification.
;; This takes the form of a list of services on its supplies and requires
;; slots. For the purposes of discussion:
;;
;;   spec_supplies    =  (get.value structure 'supplies)
;;   spec_requires    =  (get.value structure 'requires)
;;
;; All modules will have been specified correctly.
;; They will also have services recorded in supplies an requires slots.
;;
;; They also have supplies.imp and requires.imp slots.
;; These correspond to the actual services supplied and required by the
;; module. By definition these are equal to the supplies and requires spec.
;; The module.imp.rules are assumed to have been triggered during the
;; initialisation routine. They simply transfer the spec of modules
;; to the inferred .imp slots.
;;
;;
;; The data structure is introduced to help make the functions clearer.
;;
;; Each partial solution can have a number of implied properties:
;;
;;   total_supplies      the services the construction would supply
;;                    =  union of the services supplied by the modules
;;                       in the structure
;;
;;   total_requires      the services the construction would require
;;                    =  union of the services required by the modules
;;                       in the structure
;;                       -
;;                       any services that appear in total_supplies
;;
;;   total-spec_supplies  = total_supplies - spec_supplies
;;
;;   total-spec_requires  = total_requires - spec_requires
;;
;;   spec-total_supplies  = spec_supplies - total_supplies
;;
;;   spec-total_requires  = spec_requires - total_requires
;;
;; The outstanding supplies services that were specified of the
;; structure cna be seen to be
;;                        = spec-total_supplies
;;
;; The outstanding requires services that were specified of the
;; structure can be seen to be
;;                        = spec-total_requires
;;
;; The internally needed services of the partial solution
;; can be seen to be      = total-spec_requires
;;
;; Thus, the serivces that are still needed of this partial solution
;; are
;;
;;   need_to_supply        = spec-total_supplies U total-spec_requires
;;   need_to_require       = spec-total_requires
;;
;;
;; The data structure is called a state. There is one state for each
;; partial solution.
;; It takes the form:
;;
;;   state = (structure possible-solution total-spec_requires
;;                      spec-total_supplies spec-total_requires
;;                      total-spec_supplies)


;; Turn a possible solution into a state.

(defun result-of-possible-solution (structure possible-solution)
  (let*
     ( (total     (semantics-of-sup&req-f possible-solution))
       (total_supplies  (car total))
       (total_requires  (cadr total))
       (total-spec_requires  (set-difference total_requires
                                  (get.value structure 'requires)))
       (spec-total_supplies  (set-difference (get.value structure 'supplies)
                                   total_supplies))
       (spec-total_requires  (set-difference (get.value structure 'requires)
                                   total_requires))
       (total-spec_supplies  (set-difference total_supplies
                                  (get.value structure 'supplies))))

    (list structure possible-solution total-spec_requires
          spec-total_supplies spec-total_requires total-spec_supplies)))


;; I use one function to contain the semantics of the supplies and requires
;; slots when a structure is combined.

(defun semantics-of-sup&req-f (possible-solution)
  (cond ( (null possible-solution) (list nil nil))
        ( t
  (let* ( (supplies (my-union
                         (mapcar #'(lambda (c) (get.value c 'supplies.imp))
                                 possible-solution)))
          (requires (set-difference
                          (my-union
                           (mapcar #'(lambda (c) (get.value c 'requires.imp))
                                   possible-solution))
                          supplies)) )
    (list supplies requires)))))


;; Test if state is a solution or not.

(defun is-state-a-solution (state sol&hidden)
  (let ( (possible-solution (cadr state))
         (total-spec_requires (caddr state))
         (spec-total_supplies (caddr (cdr state)))
         (spec-total_requires  (caddr (cddr state)))
```

```
          (total-spec_supplies (caddr (cddr (cdr state))))
          (solutions (mapcar #'car sol&hidden)) )
     (and (not (member possible-solution solutions :test #'my-set-equal))
          (null total-spec_requires)     ; reqs no more than specified
;;        (null total-spec_supplies)     ; sups no more than specified
          (null spec-total_requires)     ; reqs no less than spec
          (null spec-total_supplies))))  ; sups no less than spec

(defun hidden-services-f (state)
  (let ( (total-spec_supplies (caddr (cddr (cdr state)))) )
;; RETURN
    total-spec_supplies))

;; Given a set of partial states,
;; check them for and collect solutions,
;; expanding search in a breadth first fashion as it goes.

(defun find-solutions (partial-states-l solutions)
  (cond ( (null partial-states-l)
;;         (print "Solutions are ")
;;         (print solutions)
           solutions)
         ( t
           (apply #'find-solutions
                  (find-solutions-do (car partial-states-l)
                                     (cdr partial-states-l)
                                     solutions)))))

(defun find-solutions-do (state rest-of-states solutions)
  (let ( (possible-solution (cadr state)) )
    (cond ( (is-state-a-solution state solutions)
            (print "found solution")
            (print (unit* possible-solution))
            (print "hidden-services")
            (print (unit* (hidden-services-f state)))
            (list rest-of-states
                  (cons (list possible-solution (hidden-services-f state))
                        solutions)) )
          ( t (list (extend-search-of-states state rest-of-states)
                    solutions))))))

;; It is in this function that I could ask user if he needs any more,
;; solutions or not.
;;
;; The search strategy could also be changed.
;; Here I use breadth first.

(defun extend-search-of-states (state rest-of-states)
  (let* ( (structure           (car state))
          (possible-solution   (cadr state))
          (total-spec_requires (caddr state))       ;; internally-needed
          (spec-total_supplies (caddr (cdr state))) ;; outstanding-supplies
          (spec-total_requires (caddr (cddr state)));; outstanding-requires
          (need_to_supply      (union spec-total_supplies
                                      total-spec_requires))
          (need_to_require     spec-total_requires) )
    (append rest-of-states
            (mapcar #'(lambda (poss-sol)
                        (result-of-possible-solution structure poss-sol))
                    (extend-partial-solution structure possible-solution
                                        need_to_supply need_to_require)))))


;; Extend partial solution in all ways possible.

(defun extend-partial-solution (structure partial-solution
                                          need_to_supply need_to_require)
  (generate-possible-solutions
        partial-solution
        (generate-possible-components structure
                                     partial-solution
                                     need_to_supply
                                     need_to_require)))

(defun generate-possible-solutions (partial-solution possible-components)
  (cond ( (null possible-components) nil)
        ( t (remove-duplicates
             (mapcar #'(lambda (c) (cons c partial-solution))
                     possible-components) :test #'my-set-equal))))


(defun generate-possible-components
    (structure partial-solution need_to_supply need_to_require)
  (cond ( (not (null need_to_require))
          (gen-poss-comps-using-req structure partial-solution
                                    need_to_require) )
        ( t
          (gen-poss-comps-using-sup structure partial-solution
                                    need_to_supply))))

(defun gen-poss-comps-using-req (structure partial-solution need_to_require)
  (let ( (c-l     (set-difference
                   (unit.descendants 'module 'member)
                   (union partial-solution
                          (get.value structure 'cannot_contain)))))
    (cond ( (null c-l) nil)
          ( t (find-possibles-r c-l need_to_require)))))


(defun find-possibles-r (c-l need_to_require)
  (cond ( (null c-l) nil)
        ( t
          (append
           (find-possibles-r-do (car c-l) need_to_require)
           (find-possibles-r (cdr c-l) need_to_require)))))

(defun find-possibles-r-do (c need_to_require)
  (let ( (c-req   (get.value c 'requires.imp)) )
    (cond ( (not (null (intersection c-req need_to_require))) (list c))
          ( t nil))))


;; Similarly for supplies

(defun gen-poss-comps-using-sup (structure partial-solution need_to_supply)
  (let ( (c-l     (set-difference
                   (unit.descendants 'module 'member)
```

```
                    (union partial-solution
                           (get.value structure 'cannot_contain)))))
    (cond ( (null c-l) nil)
          ( t (find-possibles-s c-l need_to_supply)))))


(defun find-possibles-s (c-l need_to_supply)
  (cond ( (null c-l) nil)
        ( t
          (append
           (find-possibles-s-do (car c-l) need_to_supply)
           (find-possibles-s (cdr c-l) need_to_supply)))))

(defun find-possibles-s-do (c need_to_supply)
  (let ( (c-sup   (get.value c 'supplies.imp)) )
    (cond ( (not (null (intersection c-sup need_to_supply))) (list c))
          ( t nil))))
```

D Worked Example 1 in Prolog

The information concerning known modules, structures and systems is entered as a file, which must be consulted from within Prolog, before trying to solve a problem.

```
:- dynamic structure/3.
:- dynamic module/3.
:- dynamic syst/5.

structure(unix,[ubox,ushell],[uai]).
module(ubox,[connect],[uai,rs232]).
module(ushell,[uai],[uc]).

structure(unix_tm,[unix,tm],[rs232,connect]).
module(tm,[rs232],[connect,vt100]).

structure(pseudotm,[pc,pckermit],[pcai]).
module(pc,[connect],[rs232,pcai]).
module(pckermit,[pcai,connect],[vt100]).

structure(unix_pseudotm,[unix,xover,pseudotm],[connect,rs232]).
module(xover,[rs232],[connect]).

structure(layer1,[ubox,xover,pc],[rs232]).

structure(layer2,[kserver,k,kclient],[kspro,kcpro,kpro]).
module(kserver,[uai,kpro],[kspro]).
module(k,[connect,kspro,kcpro],[kpro,file_transfer]).
module(kclient,[pcai,kpro],[kcpro]).

structure(u_pc_ft,[layer1,layer2],[uai,pcai,connect]).
```

The following shows queries for details of different modules, structures and systems present in the *KB*. The problem is then posed as a goal for Prolog to solve.

```
| ?- module(N,R,S).

N = ubox,
R = [connect],
S = [uai,rs232] ;

no
| ?- structure(N,C,H).

N = unix,
C = [ubox,ushell],
H = [uai] ;

no
| ?- syst(N,R,S,C,H).

no
| ?- system(unix,R,S,C,H).
```

```
R = [connect],
S = [rs232,uc],
C = [ubox,ushell],
H = [uai]

| ?- system(unix_tm,R,S,C,H).

R = [],
S = [vt100,uc],
C = [unix,tm],
H = [rs232,connect]

| ?- construct_system(unix,[connect],[uc,rs232],C,H).

C = [ubox,ushell],
H = [uai] ;

C = [ubox,unix_tm],
H = [uai,vt100] ;

C = [ubox,unix_pseudotm],
H = [uai,vt100] ;

C = [pseudotm,unix_tm],
H = [vt100] ;

C = [pseudotm,unix_pseudotm],
H = [vt100] ;

C = [pc,unix_tm],
H = [pcai,vt100] ;

C = [pc,unix_pseudotm],
H = [pcai,vt100] ;

Prolog interruption (h for help)? a
[ Execution aborted ]
```

The search for further solutions was stopped after 20 minutes (real time) had elapsed. The first solution was found relatively quickly though (about 5 seconds) and is the simplest correct solution.

E Z Specification

E.1 The Simplified Problem

E.1.1 Statement of Problem

Given a set of *modules*, generate a *system*, composed of a set of modules, that meets a given specification of *services supplied, Sspec*, and *required, Rspec*. Each module supplies a set of services, S, and requires a set of services, R. The semantics of composition of modules is as follows:

Given a set of modules, C, the supplied services, St, are given by:

$$St = \bigcup_{Over\ C} S$$

and the required services, Rt, given by:

$$Rt = \bigcup_{Over\ C} R - St$$

Each module can only be used once in the composition of a system. The constructed system must supply and require *exactly* those services specified *Sspec* and *Rspec*.

E.1.2 Specification of Problem

There are two basic types, a set of *modules* and a set of *services*: [*MODULE*] and [*SERVICE*]. Z1

Supplied and required services are functions of *MODULE*: Z2 Z3

$supplies : MODULE \rightarrow \mathbb{P}\ SERVICE$
$requires : MODULE \rightarrow \mathbb{P}\ SERVICE$

There is one integrity constraint that must be met by any set of data presented as a problem:

A module cannot both supply and require the same service. Z4

$$\forall m : MODULE \bullet supplies(m) \cap requires(m) = \{\}$$

The problem is stated: the aim is to construct a system composed of modules to meet specified constraints on the supplied and required services of that composition. Z5

System

$s?, r? : \mathbb{P}\ SERVICE$
$c! : \mathbb{P}\ MODULE$

$s? \cap r? = \{\}$

$\bigcup supplies(\!| c! |\!) = s?$

$\bigcup (requires(\!| c! |\!)) - s? = r?$

E.1.3 Specification of Implementation

Input specified supplied and required services of the goal system, checking integrity. Z6

```
── Inputs ──────────────────────────────
│ s?, r? : P SERVICE
├──────────
│ s? ∩ r? = {}
└──────────────────────────────────────
```

The problem-solver will attempt to find a solution by searching through a state space of partial-solutions. *Search0* describes states in that state space. Z7

```
── Search0 ─────────────────────────────
│ Inputs
│ c : P MODULE
└──────────────────────────────────────
```

The goal system does not change during the search: Z8

```
── ΔSearch0 ────────────────────────────
├──────────
│ s?′ = s?
│ r?′ = r?
└──────────────────────────────────────
```

All searches start from an initial state in the state space, in which the components in the partial solution are *nil*. Z9

```
── Init0 ───────────────────────────────
│ Search0′
├──────────
│ c′ = {}
└──────────────────────────────────────
```

A naive approach to searching the state space would be to select a module, which is not already in the components of the system, and add it to the components. Z10

```
── Step0 ───────────────────────────────
│ ΔSearch0
├──────────
│ ∃ m : MODULE • m ∉ c ∧ c′ = c ∪ {m}
└──────────────────────────────────────
```

Goal states in the search-space are those for which the components meet the constraints of being a system given the specified supplied and required services. These are solution states. Z11

```
── Solution ────────────────────────────
│ Search0₁
├──────────
│ System[c₁/c!, s?₁/s?, r?₁/r?]
└──────────────────────────────────────
```

E.1.4 Refining the Implementation

Refine *Step0*: Search does not continue once a solution-state has been reached. Add a pre-condition to *Step0*. Z12

$$\begin{array}{|l}
\textit{Step1} \\ \hline
\Delta \textit{Search0}_2 \\ \hline
\neg\ \textit{Solution}[c_2/c_1, s?_2/s?_1, r?_2/r?_1] \\
\exists\, m : \textit{MODULE} \bullet m \notin c_2 \land c'_2 = c_2 \cup \{m\} \\ \hline
\end{array}$$

Modules are only eligible for addition to components if they are not already in components and do *not* supply services required by the goal system. Z13

$$\begin{array}{|l}
\textit{Search1} \\ \hline
\textit{Search0} \\
\textit{eligible\_modules} : \mathbb{P}\ \textit{MODULE} \\ \hline
\textit{eligible\_modules} \cap c = \{\} \\ \hline
\end{array}$$

Refine *Init1*: Z14

$$\begin{array}{|l}
\textit{Init1} \\ \hline
\textit{Search1}' \\ \hline
c' = \{\} \\
\textit{eligible\_modules}' = \{em : \textit{MODULE} \mid \textit{supplies}(em) \cap r?' = \{\}\} \\ \hline
\end{array}$$

Refine *Step1*: Z15

$$\begin{array}{|l}
\textit{Step2} \\ \hline
\Delta \textit{Search1}_2 \\ \hline
\neg\ \textit{Solution}[c_2/c_1, s?_2/s?_1, r?_2/r?_1] \\
\exists\, m : \textit{MODULE} \bullet \\
\quad m = \mu(\textit{eligible\_modules}'_2 - \textit{eligible\_modules}_2) \land m = \mu(c'_2 - c_2) \\ \hline
\end{array}$$

Step2 can be implemented in 2 ways, capturing different ways of guiding the search through the state space:

1. m is chosen because it requires services that the goal system is specified to require. Z16

$$\begin{array}{|l}
\textit{Heuristic2} \\ \hline
\textit{Step2} \\ \hline
\textit{requires}(m) \cap ((r? - \bigcup \textit{requires}(\!| c_2 |\!)) - \bigcup \textit{supplies}(\!| c_2 |\!)) \neq \{\} \\ \hline
\end{array}$$

2. m is chosen because it supplies services that the the goal system is specified to supply. Z17

Heuristic3

Step2

$supplies(m) \cap (s? - \bigsqcup supplies(\!|c_2|\!)) \neq \{\}$

3. Heuristic 1: Always try to extend search by *Heuristic2* before resorting to *Heuristic3*. Z18

4. Corollary to Heuristic 1: If the outstanding required services, of the goal system, is not *nil*, and *Heuristic2* does not suggest a suitable module to add to the partial solution, then it is not possible to construct a system to meet the specification which contains this partial solution. Z19

$$\vdash \forall s?, r?, : \mathbb{P}\ SERVICE;\ c : \mathbb{P}\ MODULE \bullet$$
$$(\not\exists m : MODULE \bullet HEURISTIC2[c \cup \{m\}/c'_2] \wedge (r? - \bigsqcup requires(\!|c|\!) \neq \{\}))$$
$$\Rightarrow$$
$$\not\exists c_1 : \mathbb{P}\ MODULE \bullet (c_1 \supset c \wedge Solution[c_1/c]).$$

E.1.5 Summary of Z — by Expanding Schema

The problem: Z20

System

$s?, r? : \mathbb{P}\ SERVICE$

$c! : \mathbb{P}\ MODULE$

$s? \cap r? = \{\}$

$\bigsqcup supplies(\!|c!|\!) = s?$

$\bigsqcup(requires(\!|c!|\!)) - s? = r?$

A state in state space: Z21

Search1

$s?, r? : \mathbb{P}\ SERVICE$

$c : \mathbb{P}\ MODULE$

$eligible\_modules : \mathbb{P}\ MODULE$

$s? \cap r? = \{\}$

$eligible\_modules \cap c = \{\}$

The initial state in state space , from which to search, is set up: Z22

Init1

$s?', r?' : \mathbb{P}\, SERVICE$
$c' : \mathbb{P}\, MODULE$
$eligible\_modules' : \mathbb{P}\, MODULE$

$s?' \cap r?' = \{\}$

$eligible\_modules' \cap c' = \{\}$

$c' = \{\}$

$eligible\_modules' = \{em : MODULE \mid supplies(em) \cap r?' = \{\}\}$

A solution state is defined as: Z23

Solution

$s?_1, r?_1 : \mathbb{P}\, SERVICE$
$c_1 : \mathbb{P}\, MODULE$
$eligible\_modules : \mathbb{P}\, MODULE$

$s?_1 \cap r?_1 = \{\}$

$eligible\_modules \cap c_1 = \{\}$

$System[c_1/c!, s?_1/s?, r?_1/r?]$

A search step is defined by: Z24

Step2

$s?, r?, s?', r?' : \mathbb{P}\, SERVICE$
$c, c' : \mathbb{P}\, MODULE$
$eligible\_modules, eligible\_modules' : \mathbb{P}\, MODULE$

$s? \cap r? = \{\}$

$s?' = s?$

$r?' = r?$

$eligible\_modules \cap c = \{\}$

$eligible\_modules' \cap c' = \{\}$

$\neg\, Solution[c/c_1, s?/s?_1, r?/r?_1]$

$\exists\, m : MODULE \bullet$
$\quad m = \mu(eligible\_modules' - eligible\_modules) \land m = \mu(c' - c)$

The heuristics for search are: Z25

Heuristic2

$$\begin{array}{l}
s?, r?, s?', r?' : \mathbb{P}\, SERVICE \\
c, c' : \mathbb{P}\, MODULE \\
eligible\_modules, eligible\_modules' : \mathbb{P}\, MODULE \\
\hline
s? \cap r? = \{\} \\
s?' = s? \\
r?' = r? \\
eligible\_modules \cap c = \{\} \\
eligible\_modules' \cap c' = \{\} \\
\neg\, Solution[c/c_1, s?/s?_1, r?/r?_1] \\
\exists\, m : MODULE \bullet \\
\quad m = \mu(eligible\_modules' - eligible\_modules) \land m = \mu(c' - c) \\
requires(m) \cap ((r? - \bigsqcup requires(\!| c |\!)) - \bigsqcup supplies(\!| c |\!)) \neq \{\}
\end{array}$$

Z26

Heuristic3

$$\begin{array}{l}
s?, r?, s?', r?' : \mathbb{P}\, SERVICE \\
c, c' : \mathbb{P}\, MODULE \\
eligible\_modules, eligible\_modules' : \mathbb{P}\, MODULE \\
\hline
s? \cap r? = \{\} \\
s?' = s? \\
r?' = r? \\
eligible\_modules \cap c = \{\} \\
eligible\_modules' \cap c' = \{\} \\
\neg\, Solution[c/c_1, s?/s?_1, r?/r?_1] \\
\exists\, m : MODULE \bullet \\
\quad m = \mu(eligible\_modules' - eligible\_modules) \land m = \mu(c' - c) \\
supplies(m) \cap (s? - \bigsqcup supplies(\!| c |\!)) \neq \{\}
\end{array}$$

E.2 Rules

```
(init1
    (if (the goal of ?s is solve)
        (find.any (?s is in class system))
        (?ms = (lisp (eligible_modules-f nil (requires-f ?s))))
        then
        in.new.world
        (add (the components of ?s is nil) using no.rules)
        (add (the eligible_modules of ?s is ?ms) using no.rules)
        (change.to (the goal of ?s is search))))

(heuristic2
   (if (the goal of ?s is search)
       (the components of ?s is ?c)
       (the eligible_modules of ?s is ?em)
       (?m is in class module)
       (lisp (member ?m ?em))
       (not (lisp (member ?m ?c)))
       (lisp
         (not
           (null
             (intersection
                 (requires-f ?m)
                 (set-difference
                     (set-difference
                         (the requires of ?s)
                         (union* (mapcar #'requires-f ?c)))
                     (union* (mapcar #'supplies-f ?c)))))))
       (?new_em = (lisp (set-difference ?em (list ?m))))
       (?c_new  = (lisp (cons ?m ?c)))
       (for
         (true.in.world
           ((the goal of ?s is solved) and
    (the components of ?s is ?c_old)) ?w)
         always
         (lisp (not (subsetp ?c_old ?c_new))))
       then
       in.new.world
       (change.to (the eligible_modules of ?s is ?new_em)
using no.rules)
       (change.to (the components of ?s is ?c_new))))

(heuristic3
    (if (the goal of ?s is search)
        (the components of ?s is ?c)
        (the eligible_modules of ?s is ?em)
        (?m is in class module)
        (lisp (member ?m ?em))
        (not (lisp (member ?m ?c)))
        (lisp
          (not
            (null
              (intersection
(supplies-f ?m)
                (set-difference
(the supplies of ?s)
                       (union* (mapcar #'supplies-f ?c)))))))
        (?new_em = (lisp (set-difference ?em (list ?m))))
```

```
          (?c_new  = (lisp (cons ?m ?c)))
          (for
            (true.in.world
              ((the goal of ?s is solved) and
       (the components of ?s is ?c_old)) ?w)
             always
            (lisp (not (subsetp ?c_old ?c_new))))
          then
          in.new.world
          (change.to (the eligible_modules of ?s is ?new_em)
using no.rules)
          (change.to (the components of ?s is ?c_new))))

(solution
    (if (the components of ?s is ?c)
        (the supplies of ?s is ?sup)
        (the requires of ?s is ?req)
        (lisp (system-p ?req ?sup ?c))
        then
        (change.to (the goal of ?s is solved))))
```

E.3 Lisp

E.3.1 Lisp Functions

```
(in-package 'kee)

;; Define supplies-f and requires-f to meet Z specification.

(defun supplies-f (module)
  (get.value module 'supplies))

(defun requires-f (module)
  (get.value module 'requires))


;; Predicate to implement SYSTEM schema

(defun system-p (req sup comp)
  (and (null (intersection req sup))
       (my-set-equal sup (union* (mapcar #'supplies-f comp)))
       (my-set-equal req (set-difference
                           (union* (mapcar #'requires-f comp)) sup))))


;; eligible_modules-f
;;
;; Is a function from the components and requires of a system,
;; plus the database of modules,
;; to those modules that are eligible for adding to components.

(defun eligible_modules-f (comp requires)
  (letbindings ( (?em nil)
                 (?comp comp))
    (loop.with.bindings (?m is in class module)
        when (null (intersection (supplies-f ?m) requires))
```

```
                                        (lisp (not (member ?m ?comp)))
                                        do (setq ?em (cons ?m ?em)))
    ;; Return
      ?em))
```

E.3.2 Sets in Lisp

```
(in-package 'kee)

;; set operators

(defun my-set-equal (s1 s2)
  (and (null (set-difference s1 s2))
       (null (set-difference s2 s1))))

(defun my-intersection (l1)
  (cond ((null l1) nil)
        ((null (cdr l1)) (car l1))
        (t
         (my-intersection
          (cons
           (intersection (car l1) (cadr l1) :test #'equal)
           (cddr l1)))))))

(defun my-union (l1)
  (cond ((null l1) nil)
        ((null (cdr l1)) (car l1))
        (t
         (my-union
          (cons
           (union (car l1) (cadr l1) :test #'equal)
           (cddr l1)))))))

(defun union* (l1) (my-union l1))

(defun my-set-difference (l1 l2)
  (set-difference l1 l2 :test #'equal))
```

Bibliography

BRACHMAN, R J & H J LEVESQUE, (eds.) 1985. *Readings in Knowledge Representation.* Morgan Kaufmann. A collection of seminal papers.

BRACHMAN, R J & J G SCHMOLZE 1985. "An Overview of the KL-ONE Knowledge Representation System", *Cognitive Science,* 9: 171–216.

CHARNIAK, EUGENE & DREW MCDERMOTT 1985. *Introduction to Artificial Intelligence.* Addison-Wesley Publishing Company.

DEKLEER, J 1986. *Artificial Intelligence Journal,* 28 (2).

FROST, R A 1986. *Introduction to Knowledge Base Systems.* Collins.

HAYES, IAN, (ed.) 1987. *Specification Case Studies.* Prentice-Hall International (UK) Ltd.

HENDERSON, PETER & BRIAN WARBOYS 1989a. "Alfa — A Language for Configuration Description", Report, University of Southampton and University of Manchester. July.

HENDERSON, PETER & BRIAN WARBOYS 1989b. "Alfa — A Language for Configuration Description", Report, University of Southampton and University of Manchester. December.

HENDERSON, PETER & BRIAN WARBOYS 1989c. "An Architectural Framework for Systems", ICL Technical Journal.

JACKENDOFF, R 1983. *Semantics and Cognition.* MIT Press.

LEHLER, WM 1988. *Constraint Propagation Languages.* Addison-Wesley.

MCCLUSKEY, T L 1988. "Deriving a correct logic program from the formal specification of a non-linear planner". *in* Rais & Saitti, (eds.), *Methodologies for Intelligent Systems.* North-Holland.

MINSKY, MARVIN 1975. "A Framework for Representing Knowledge". *in* Winston, P. H., (ed.), *The Psychology of Computer Vision,* pp. 211–280. McGraw-Hill.

O'SHEA, TIM & MARC EISENSTADT, (eds.) 1984. *Artificial Intelligence.* Harper and Row, New York.

QUILLIAN, M ROSS 1966. "Semantic Memory", Report AD-6416721, Clearinghouse for Federal Scientific and Technical Information. Abridged version in M Minsky (1968), Semantic Information Processing, MIT Press, Cambridge, MA.

ROBERTS, DON D 1973. *The Existential Graphs of Charles S Peirce.* Mouton, University of Waterloo.

SCHAFER, G 1976. *A Mathematical Theory of Evidence.* Princeton University Press.

SEARLE, JOHN 1980. "Minds, Brains and Programs", *Behavioural and Brain Sciences Journal*, 3: 417–8.

SHADBOLT, N & M BURTON 1989. "Knowledge Elicitation". *in* Wilson, J. & Corlett, N., (eds.), *Evaluation of Human Work: Practical Ergonomics Methodology.* Taylor and France.

SHAPIRO, S C, (ed.) 1987. *Encyclopaedia of Artificial Intelligence, Volumes 1 and 2.* John Wiley and Sons. ISBN 0-471-80748-6 (set).

SHORTLIFFE, E H & B G BUCHANAN 1975. "A Model of Inexact Reasoning in Medicine", *Mathematical Biosciences*, 23: 351–379.

SMITH, BRIAN C 1985. "Prologue to 'Reflection and Semantics in a Procedural Language'". *in* Brachman, R. J. & Levesque, H. J., (eds.), *Readings in Knowledge Representation.* Morgan Kaufmann Publishers, Inc.

SOWA, JOHN F 1968. "Conceptual Structures", Unpublished manuscript.

SOWA, J F 1976. "Conceptual Graphs for a Data Base Interface", *IBM Journal of Research and Developments.*

SOWA, J F 1984. *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley.

SOWA, JOHN F & EILEEN C WAY 1986. "Implementing a Semantic Interpreter using Conceptual Graphs", *IBM Journal of Research Developments*, 30 (1).

STEELE, GUY L 1984. *Common LISP.* Digital Press.

TURING, A M October 1950. "Computing Machinery and Intelligence", *Mind*, 59 (236): 433–460.

WINOGRAD, T 1972. *Understanding Natural Language.* Academic Press, New York.

WINSTON, P H 1984. *Artificial Intelligence (Second Edition).* Addison-Wesley.